RTA-OSEK

User Guide

Contact Details

ETAS Group

www.etasgroup.com

Germany

ETAS GmbH
Borsigstraße 14
70469 Stuttgart

Tel.:+49 (711) 8 96 61-102
Fax:+49 (711) 8 96 61-106

www.etas.de

USA

ETAS Inc.
3021 Miller Road
Ann Arbor, MI 48103

Tel.: +1 (888) ETAS INC
Fax: +1 (734) 997-94 49

www.etasinc.com

Japan

ETAS K.K.
Queen's Tower C-17F,
2-3-5, Minatomirai, Nishi-ku,
Yokohama, Kanagawa
220-6217 Japan

Tel.: +81 (45) 222-0900
Fax: +81 (45) 222-0956

www.etas.co.jp

France

ETAS S.A.S.
1, place des États-Unis
SILIC 307
94588 Rungis Cedex

Tel.: +33 (1) 56 70 00 50
Fax: +33 (1) 56 70 00 51

www.etas.fr

Korea

ETAS Korea Co. Ltd.
3F, Samseung Bldg. 61-1
Yangjae-dong, Seocho-gu
Seoul

Tel.: +82 (2) 57 47-016
Fax: +82 (2) 57 47-120

www.etas.co.kr

Great Britain

ETAS UK Ltd.
Studio 3, Waterside Court
Third Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ

Tel.: +44 (0) 1283 - 54 65 12
Fax: +44 (0) 1283 - 54 87 67

www.etas-uk.net

Copyright Notice

© 2001 - 2004 LiveDevices Ltd. All rights reserved.

Version: RM00005-005

No part of this document may be reproduced without the prior written consent of LiveDevices Ltd. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

Disclaimer

The information in this document is subject to change without notice and does not represent a commitment on any part of LiveDevices. While the information contained herein is assumed to be accurate, LiveDevices assumes no responsibility for any errors or omissions.

In no event shall LiveDevices, its employees, its contractors or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees or expenses of any nature or kind.

Trademarks

RTA-OSEK and LiveDevices are trademarks of LiveDevices Ltd.

Windows and MS-DOS are trademarks of Microsoft Corp.

OSEK/VDX is a trademark of Siemens AG.

All other product names are trademarks or registered trademarks of their respective owners.

Certain aspects of the technology described in this guide are the subject of the following patent applications:

UK - 0209479.5 and USA - 10/146,654,

UK - 0209800.2 and USA - 10/146,239,

UK - 0219936.2 and USA - 10/242,482.

Contents

1 About this Guide 1-1
1.1 Who Should Read this Guide? 1-1
1.2 Conventions 1-1
1.2.1 Screenshots 1-2

2 Introduction 2-1
2.1 RTA-OSEK Component 2-2
2.2 RTA-OSEK tools 2-2
2.3 RTA-OSEK debugging support 2-3
2.4 OSEK 2-4
2.5 New Features in RTA-OSEK 4.0 2-4
2.5.1 Importing Files 2-4
2.5.2 Auxiliary OIL files 2-4

3 The Development Process 3-1
3.1 Overview 3-1
3.1.1 Specification 3-2
3.1.2 Implementation 3-3

3.1.3 Build 3-5
3.1.4 Functional Testing 3-5
3.1.5 Timing Analysis 3-6
3.2 A Simple Example 3-6
3.2.1 Creating a New Application using the RTA-OSEK GUI 3-6
3.2.2 Saving the Application 3-8
3.2.3 Viewing the OIL File 3-8
3.2.4 Implementation 3-9
3.2.5 Build 3-24
3.2.6 Functional Testing 3-24
3.3 A Simple Example Using Timing Analysis 3-24
3.3.1 Your Specification 3-24
3.3.2 Implementation 3-36
3.3.3 Build 3-52
3.3.4 Functional Testing 3-53
3.3.5 Analysis 3-53
3.4 Completion of the Examples 3-62
3.5 RTA-OSEK *Builder* 3-63
3.5.1 Basic Data Entry 3-63
3.5.2 Building an application 3-63
3.5.3 Build Checks 3-64
3.5.4 Create Files 3-64
3.5.5 Custom Build 3-66
3.5.6 Custom Build Options 3-66
3.6 Other Implementation Details 3-71
3.6.1 Namespace 3-71
3.6.2 Reentrancy 3-71

4 Tasks 4-1
4.1 Task Switching 4-1
4.2 Basic and Extended Tasks 4-2
4.2.1 Basic Tasks 4-2
4.2.2 Extended Tasks 4-4
4.3 Simple Task Configuration 4-6
4.4 Task Entry 4-7

4.5 Task Termination 4-8
4.5.1 API Calls for Task Termination 4-8
4.5.2 Heavyweight and Lightweight Termination 4-8
4.6 Simple Task Activation 4-11
4.7 The Idle Task 4-12
4.8 Multiple Activation 4-13
4.9 Non-Preemptive Tasks 4-14
4.9.1 Rescheduling 4-14
4.10 Fast Activation 4-15
4.11 Floating-Point 4-16
4.11.1 Customizing Floating-Point Operation 4-17
4.12 Advanced Task Activation 4-17
4.12.1 Direct Activation 4-17
4.12.2 Activation by Event 4-18
4.12.3 Activation by a Message 4-18
4.12.4 Activation by an Alarm 4-18
4.12.5 Activation by a Periodic Schedule 4-18
4.12.6 Activation by a Planned Schedule 4-18
4.12.7 Auto-Activation 4-19
4.13 Tasksets 4-19
4.13.1 Predefined Tasksets 4-20
4.14 Choice of Task Characteristics 4-20
4.15 Controlling Task Execution Ordering 4-21
4.15.1 Direct Activation Chains 4-21
4.15.2 Using Priority Levels 4-22
4.16 Synchronization with Basic Tasks 4-23
4.17 Simulating Waiting using Basic Tasks 4-24

5 Interrupts 5-1
5.1 Single-Level and Multi-Level Platforms 5-1
5.2 Interrupt Service Routines 5-1
5.2.1 Category 1 and Category 2 Interrupts 5-2
5.3 Interrupt Priorities 5-3
5.3.1 OS Level 5-4
5.3.2 User Level 5-4

5.4 Simple Interrupt Configuration 5-5
5.5 Providing Interrupt Handlers 5-6
5.5.1 Category 1 Interrupt Handlers 5-6
5.5.2 Category 2 Interrupt Handlers 5-7
5.6 Enabling and Disabling Interrupts 5-8
5.7 Getting Resources 5-9
5.8 Interrupt Arbitration 5-9
5.9 Using Floating-Point 5-10
5.10 The Default Interrupt 5-11
5.11 Shortening Interrupt Response Times 5-12

6 Resources 6-1
6.1 Resource Configuration 6-1
6.2 Getting and Releasing Resources 6-2
6.2.1 Nesting Resource Calls 6-3
6.3 Using the Static Interface 6-4
6.4 Combined Resources 6-4
6.5 Linked Resources 6-5
6.6 Internal Resources 6-6
6.7 Using the Scheduler as a Resource 6-8
6.8 Choosing a Pre-Emption Control Mechanism 6-8
6.9 Avoiding Race Conditions 6-9

7 Events 7-1
7.1 Configuring Events 7-1
7.2 Waiting on an Event 7-3
7.3 Setting Events 7-4
7.4 Clearing Events 7-4
7.5 Events and the Idle Task 7-5
7.6 Waiting On, Setting and Clearing Multiple Events 7-5
7.7 Advanced Event Setting 7-6
7.7.1 Setting Events with an Alarm 7-6
7.7.2 Setting Events with a Message 7-6

8 Messages 8-1
8.1 Communication in OSEK 8-1
8.2 Configuring Messages 8-1

8.2.1 Declaring Messages 8-1
8.2.2 Declaring Senders and Receivers 8-4
8.2.3 Specifying Accessors 8-4
8.2.4 Specifying Transmission Mechanisms 8-6
8.3 Sending and Receiving Messages 8-8
8.3.1 Sending a Message 8-8
8.3.2 Receiving a Message 8-9
8.4 Starting and Stopping COM 8-10
8.5 Initialization and Shutdown of COM 8-10
8.6 Queued Messages 8-11
8.7 Mixed-Mode Transmission 8-12
8.8 Activating Tasks on Message Transmission 8-12
8.9 Setting Events on Message Transmission 8-12
8.10 Callback Routines 8-13
8.11 Using Flags 8-14

9 Introduction to System Modeling 9-1
9.1 Declaring Stimuli and Responses 9-1
9.2 Arrival Patterns and Arrival Rates 9-2
9.3 Designing Responses 9-3
9.4 Designing Stimuli 9-4

10 Counters and Alarms 10-1
10.1 The Counter/Alarm Mechanism 10-1
10.2 Declaring Counters 10-2
10.3 Declaring Alarms 10-2
10.3.1 Alarm Actions 10-4
10.3.2 Autostarting Alarms 10-5
10.4 Incrementing Counters 10-5
10.5 Setting Alarms 10-6
10.5.1 Alarm Examples 10-7
10.6 Canceling Alarms 10-9
10.7 Alarm Callbacks 10-9
10.8 Setting an Event 10-10
10.9 Releasing Multiple Tasks from Alarms 10-10
10.10 Changing Counter Attributes 10-10

10.11 Using Non-Time Based Counter Units 10-11
10.12 Specifying Counter Constants 10-12
10.13 Determining the Next Alarm Expiry 10-13
10.14 Autostarting Alarms 10-13
10.15 Real-time and CPU-time 10-13
10.16 Synchronization using Alarms 10-14
10.17 Absolute Periodic Alarms 10-15
10.18 Aperiodic Alarms 10-15

11 Schedules 11-1
11.1 Using Schedules 11-1
11.1.1 Types of Schedules 11-1
11.1.2 Arrivalpoints 11-1
11.1.3 Ticked and Advanced Schedules 11-3
11.2 Declaring Periodic Schedules 11-3
11.3 Building Periodic Schedules 11-4
11.3.1 Visualization of a Periodic Schedule 11-5
11.3.2 Editing Periods 11-6
11.4 Declaring Planned Schedules 11-6
11.5 Building Planned Schedules 11-7
11.5.1 Planning Arrivalpoints 11-8
11.5.2 Attaching Stimuli to Arrivalpoints 11-9
11.5.3 Visualization of a Planned Schedule 11-9
11.5.4 Editing Plans 11-10
11.6 Starting Schedules 11-10
11.7 Ticking Schedules 11-11
11.8 Stopping Schedules 11-12
11.9 Autostarting Ticked Schedules 11-12
11.10 Restarting Single-Shot Schedules 11-13
11.11 Advancing Schedules 11-13
11.11.1 Advanced Schedule Driver Callbacks 11-15
11.12 Using Non-Time Based Schedule Units 11-16
11.13 Specifying Schedule Constants 11-16
11.14 Using Periodic Offsets 11-17
11.15 Modifying Planned Schedules at Run-time 11-19

11.15.1 Modifying Delays 11-20
11.15.2 Modifying Next Values 11-20
11.15.3 Modifying Auto-Activated Tasks 11-21
11.16 Schedule/Arrivalpoint Tradeoffs 11-22
11.17 Minimizing Schedule RAM Usage 11-22

12 Startup and Shutdown 12-1
12.1 From System Reset to `StartOS()` 12-1
12.1.1 Power-on or Reset to `main()` 12-1
12.1.2 The Application Start-up Code 12-2
12.1.3 Memory Images and Linker Files 12-5
12.1.4 Downloading to your Target 12-9
12.1.5 ROMability 12-9
12.2 Starting RTA-OSEK Component 12-9
12.2.1 Shutting Down RTA-OSEK Component 12-10
12.2.2 Application Modes 12-10
12.2.3 Autostarting Tasks 12-11
12.2.4 Autostarting Alarms 12-12

13 Error Handling and Execution Monitoring 13-1
13.1 Enabling Hook Routines 13-1
13.2 Mandatory Hooks 13-2
13.3 Startup Hook 13-3
13.4 Shutdown Hook 13-3
13.5 Pre and Post Task Hooks 13-4
13.6 The Error Hook 13-4
13.7 The Stack Fault Hook 13-5
13.8 Advanced Error Logging 13-6
13.8.1 Configuring Advanced Error Logging 13-6
13.8.2 Using Advanced Error Logging 13-7
13.9 Measuring and Monitoring Execution Time 13-9
13.9.1 Enabling Timing Measurement 13-9
13.9.2 Measuring Execution Times 13-10
13.9.3 Setting Timing Budgets 13-10
13.9.4 Obtaining Blocking Times 13-11
13.10 Measuring and Monitoring Stack Use 13-12

13.11 Catching Errors at Compile Time 13-14
13.12 Run-time Fault Tolerance 13-15
13.13 Imprecise Computation 13-15

14 Modeling the Application for the RTA-OSEK Planner 14-1
14.1 Configuring Applications for Analysis 14-4
14.2 Defining Stimulus/Response Timing Relationships 14-4
14.2.1 Stimulus Arrival Types and Patterns Revisited 14-4
14.2.2 Bursty Arrival Patterns 14-5
14.2.3 Periodic Arrival Patterns 14-7
14.2.4 Planned Arrival Patterns 14-8
14.2.5 Setting Deadlines for Responses 14-8
14.2.6 Specifying Response Generation Time 14-9
14.3 Capturing Execution Information 14-10
14.3.1 Primary and Activated Profiles 14-11
14.3.2 Tasks and ISRs 14-12
14.3.3 Modeling the Idle Task 14-13
14.3.4 Resource and Interrupt Locks 14-13
14.4 Target Specific Timing Information 14-16
14.4.1 System Timings 14-17
14.4.2 Interrupt Recognition Time 14-17
14.4.3 Interrupt Arbitration 14-18
14.5 Analyzing Alarms 14-19
14.6 Specifying Multiple Execution Profiles 14-20
14.7 Looping and Retriggering Interrupt Behavior 14-21
14.8 Modeling Jitter 14-24
14.9 Modeling Planned Schedules 14-26
14.9.1 Specifying Analysis Overrides 14-27
14.9.2 Indirectly Activated Stimuli 14-27
14.10 Modeling Single-Shot Schedules 14-28
14.11 Modeling with Extended Tasks 14-29

15 Performing Analysis 15-1
15.1 Stack Depth Analysis 15-1
15.1.1 Floating-Point Context Saving 15-4
15.1.2 Minimizing Stack Usage 15-4

15.2 Schedulability Analysis 15-4
15.2.1 Unschedulable Systems 15-7
15.2.2 Indeterminate Schedulability 15-11
15.3 Sensitivity Analysis 15-12
15.3.1 Sensitivity to Clock Speed 15-14
15.3.2 Sensitivity to Execution Times 15-15
15.3.3 Sensitivity to Deadlines 15-16
15.4 Best Task Priorities Analysis 15-16
15.4.1 Required Lower Priority Tasks 15-19
15.5 CPU Clock Rate Optimization 15-19

16 Using RTA-OSEK from the Command Line 16-1
16.1 Overview of Operation 16-1
16.1.1 Functionality 16-1
16.1.2 Messages 16-1
16.1.3 Return Values 16-2
16.1.4 Command Line Options 16-2
16.1.5 Output Files 16-4

17 Using RTA-OSEK with RTA-TRACE 17-1
17.1 Configuration 17-1
17.2 Tracepoints 17-4
17.3 Task Tracepoints 17-5
17.4 Intervals 17-5
17.5 Categories 17-5
17.6 Enumerations 17-5
17.7 Filters 17-5
17.8 Format Strings 17-6
17.8.1 Rules 17-6
17.8.2 Examples 17-8

1 About this Guide

This guide provides you with an introduction to RTA-OSEK. It describes the basic system concepts and shows you how to put these concepts into practice.

You will find the complete technical details of RTA-OSEK Component in the *RTA-OSEK Reference Guide*. These manuals describe the parts of RTA-OSEK that apply to all target hardware. If you require information on target-specific aspects of RTA-OSEK, refer to the supplied *RTA-OSEK Binding Manual*.

1.1 Who Should Read this Guide?

It is assumed that you are a system designer who wants to know how to model your system architecture using the RTA-OSEK GUI or that you are a C programmer who wants to know how to configure RTA-OSEK Component for integration with your application program.

1.2 Conventions

Important: Notes that appear like this contain important information that you need to be aware of. Make sure that you read them carefully and that you follow any instructions that you are given.

Portability: Notes that appear like this describe things that you will need to know if you want to write code that will work on any processor running RTA-OSEK Component.

The following terms are used in this guide:

RTA-OSEK	refers to the complete Real-Time Operating System product including the tools that run on the host PC, the target processor components and the documentation.
Offline tools	refers to the configuration, analysis and build tools that are run on the host PC. These include the RTA-OSEK graphical user interface (GUI) that provides a wrapper around the other offline tools.
RTA-OSEK GUI	refers to the RTA-OSEK graphical user interface (GUI) that provides a wrapper around the other offline tools.
RTA-OSEK Component	refers to the RTA-OSEK Real-Time Operating System kernel that runs on the target processor. Any references to *the kernel* in this guide refer to RTA-OSEK Component.

In this guide you'll see that program code, header file names, C type names, C functions and RTA-OSEK API call names all appear in the `courier` typeface. When the name of an object is made available to the programmer the name also appears in the `courier` typeface, so, for example, a task named Task1 appears as a task handle called `Task1`.

1.2.1 Screenshots

Please note that due to LiveDevices policy of continual product improvement, some of the screenshots reproduced in this manual may not exactly match the onscreen appearance of the GUI tool. GUI appearance may also be affected by your local Windows setup.

2 Introduction

The core of RTA-OSEK consists of two main elements. These elements are:

- **The RTA-OSEK offline tools**.
  The RTA-OSEK offline tools include a code generation tool and an analysis tool that enables you to demonstrate that your system meets its timing requirements. These offline tools are driven by a graphical user interface (GUI) which supports the OSEK Implementation Language (OIL) configuration. You can find out more about the RTA-OSEK offline tools in Section 2.2 and OSEK is introduced in section 2.4.
- **RTA-OSEK Component** - the OSEK kernel.
  RTA-OSEK Component is an efficient, fast and predictable Real-Time Operating System (RTOS) that is fully compliant and independently certified with Version 2.2 of the OSEK/VDX OS Standard. RTA-OSEK Component has been designed to provide the necessary functions for building complex, yet efficient, real-time systems. You can find out more about RTA-OSEK Component in Section 2.1.

RTA-OSEK supports the development of hard real-time systems. This means that system responses must be made within specific timing deadlines. Meeting hard deadlines involves calculating the worst-case response time of each task and Interrupt Service Routine (ISR) and ensuring that everything runs on time, every time.

Any true RTOS must support these requirements by meeting the assumptions of fixed priority schedulability analysis*. RTA-OSEK Component meets these requirements and the RTA-OSEK offline tools automate the analysis to show whether deadlines will be met.

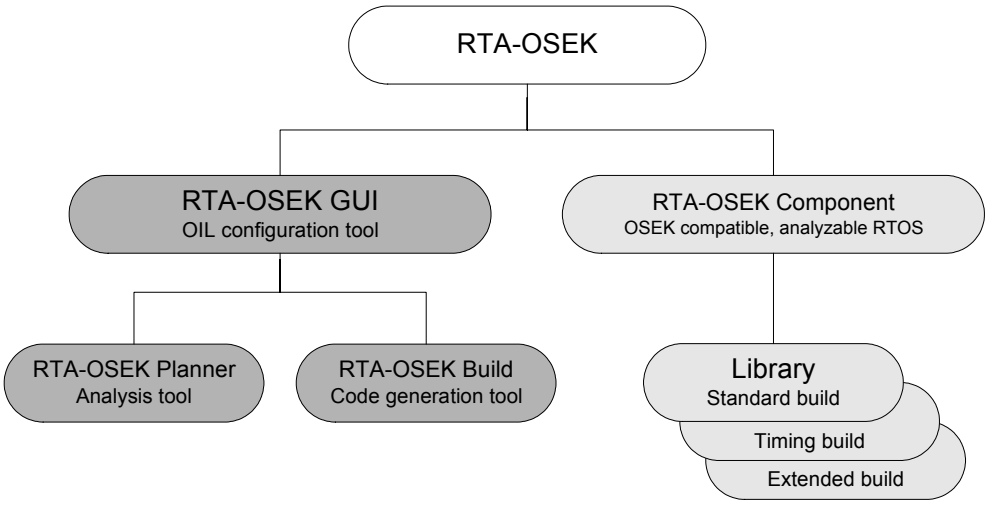


Figure 2:1 - The Components of RTA-OSEK

* For further information refer to: N.C. Audsley, A. Burns, R. I. Davis, K.W. Tindell, and A.J. Wellings, 1995 "Fixed Priority Pre-emptive Scheduling: An Historical Perspective" *Real-Time Systems*, 8, 173-198.

2.1 RTA-OSEK Component

The concepts behind RTA-OSEK Component are founded on the results of a decade of research into real-time systems and are shaped by the pressures of mass-production industries, such as the automotive industry.

RTA-OSEK Component is particularly suitable for systems manufactured in large quantities, where it is necessary to meet very tight constraints on hardware costs and where any final product must function correctly.

RTA-OSEK Component is certified to the OSEK OS Standard Version 2.2 and is statically configured using the OIL configuration language. RTA-OSEK offers support for a wide variety of microcontrollers and leads the class in its low memory footprint and CPU overhead.

Using **static API optimization** reduces the execution time of critical higher priority tasks, which means that the useable processing power is increased.

A **Timing status** build is added to the Standard and Extended status builds defined by the OSEK OS specification. This allows you to measure the worst-case execution time of tasks and to perform **execution time monitoring** (ensuring that tasks complete within specified times).

RTA-OSEK provides a number of **kernel optimizations** that contribute to reductions in unit cost of systems. The **lightweight tasks optimization**, for example, leads to RAM savings of up to 256 bytes of RAM per task. This results in substantial savings in a 32 task system.

All kernel CPU overheads (such as switching to and from tasks, handling interrupts and waking up tasks) have **low worst-case bounds** and little variability within execution times. Conventional RTOS designs normally have unpredictable overheads, usually dependent on the number of tasks and the state of the system at each point in time.

Unlike the conventional RTOS 'infinite loop' tasking (where tasks are not required to terminate), the single-shot execution model of RTA-OSEK basic tasks is an exact fit with the tasking model used in schedulability analysis. RTA-OSEK Component's **built-in execution time measurement functions** exploit this to measure task and ISR execution that can be used directly in analysis.

RTA-OSEK Component does not impose on hardware, where possible. Generally, there is no need to 'hand over' control of hardware, such as the cache, watchdog timers and I/O ports. As a result of this hardware can be used freely, allowing 'legacy software' to be brought to the system.

RTA-OSEK Component supports all four OSEK conformance classes (BCC1, BCC2, ECC1 and ECC2). It also provides message handling for intra-processor communication that satisfies the OSEK COM CCCA and CCCB conformance class.

2.2 RTA-OSEK tools

When the correct functioning of an application depends upon performance requirements (such as, how quickly responses are produced), it is often extremely difficult to guarantee that these requirements have been met. RTA-

OSEK is currently the only RTOS product on the market that allows such performance requirements to be guaranteed.

An **advanced graphical user interface** is made available to improve the development process. This interface provides **implementation obligations**, which act as a checklist for developing source code to work with the architecture defined by the OIL file.

The advanced interface also supports **stack usage analysis**. This means that you can determine the worst-case stack requirements for the application, avoiding the need to over-engineer "just in case" RAM requirements are incorrect.

RTA-OSEK not only allows you to reap the benefits of a small, fast OSEK OS with guaranteed timing behavior, but it also supports **priority level optimization**. You can use this to determine whether the preemption patterns of the system can be adjusted to reduce stack usage.

Research has shown that even where systems are running at 99% CPU utilization, this technique can be used to modify preemption patterns, which can result in an 8-fold decrease in application stack requirements. Significant RAM reductions may be made and this can lead to reduced unit costs.

RTA-OSEK also includes **sensitivity analysis**. This can assist you in determining the possibility that the execution time tasks or interrupts can be extended. This is an invaluable aid when extending or enhancing a system, without violating its performance requirements.

Clock speed minimization is a further type of analysis that is provided by RTA-OSEK. This can be used to show the slowest speed that the application can run and still meet its deadlines. You can use this functionality to reduce power requirements, to avoid EMC problems or to determine whether cheaper silicon can be used to meet the same performance requirements.

You can also use **schedulability analysis**, which is a mathematical technique used to prove that an application meets all of its deadlines. RTA-OSEK provides extensions to schedulability analysis that allow you to determine the maximum buffer sizes required by interrupts. You can use this to guide hardware selection and to determine the maximum activation count for BCC2 tasks.

RTA-OSEK also provides the ability to create and manipulate planned and periodic **schedules**. Schedules are a mechanism for managing activation of multiple tasks.

2.3 RTA-OSEK debugging support

RTA-OSEK provides **support for the ORTI** (OSEK Run-Time Interface) standard. This allows any ORTI-aware debugger to provide access to RTOS variables.

2.4 OSEK

OSEK is a European automotive industry standards effort to produce open systems interfaces for vehicle electronics. The full name of the project is OSEK/VDX.

OSEK is an acronym formed from a phrase in German, which translates as "Open Systems and Corresponding Interfaces for Automotive Electronics". VDX is based on a French standard (Vehicle Distributed eXecutive), which has now been merged with OSEK. OSEK/VDX is referred to as OSEK in this guide.

The goals of OSEK are to support portability and reusability of software components across a number of projects. This will allow vendors to specialize in "Automotive Intellectual Property", where a vendor can develop a purely-software solution and run software in any OSEK-compliant ECU.

To reach this goal, however, detailed specifications of the interfaces to each non application-specific component are required. OSEK standards, therefore, include an Application Programming Interface (API) that abstracts away from the specific details of the underlying hardware and the configuration of the in-vehicle networks.

2.5 New Features in RTA-OSEK 4.0

2.5.1 Importing Files

RTA-OSEK can merge the content of an external OIL file into the current project by importing the external file. (Menu option **File** / **Import**)

The following points should be noted:

- Imported OIL files should be syntactically 'complete'. i.e. there must be a CPU clause around the subsystem declarations.
- Settings in imported OIL files will override values previously set in the project file.
- When RTA-OSEK saves the project file, any values that originated from an imported file get saved too. If you have a subsystem that adds or removes objects such as tasks depending upon the configuration take care not to save the project file. If you do, then you may have to use the GUI to remove the objects that are no longer required. Alternatively you can specify the file as an **auxiliary OIL file** in which case the file will get imported each time the project is loaded.

2.5.2 Auxiliary OIL files

When RTA-OSEK saves a project file, it writes the complete set of configuration data to a single .oil file. If the original .oil file that was read into RTA-OSEK was composed from separate OIL file fragments bound together via the #include mechanism, this structure is lost.

Often this is what is desired. However there may be situations where some external tool is being used to maintain portions of the overall application (e.g.

a TCP/IP stack), and that tool generates a file containing OIL declarations relating to the subsystem.

If the content of the subsystem is changed, RTA-OSEK must update the project by re-reading the relevant OIL file fragment.

This can be done manually by importing the file (Menu option **File** / **Import**), or alternatively the name of the file can be added to the project as an **auxiliary OIL file**.

Auxiliary OIL files are read by RTA-OSEK after reading the main project file. They act similarly to using #include statements at the end of the project file.

The following points should be noted:

- Auxiliary OIL files should be syntactically 'complete'. i.e. there must be a CPU clause around the subsystem declarations.
- Settings in auxiliary OIL files will override values previously set in the project file or any other auxiliary OIL files that are read before the current one.
- When RTA-OSEK saves the project file, any values that originated from an auxiliary or imported file get saved too. If you have a subsystem that adds or removes objects such as tasks depending upon the configuration take care not to save the project file. If you do, then you may have to use the GUI to remove the objects that are no longer required.

Further information can be found in the online-help.

3 The Development Process

This chapter will guide you through the processes involved in creating an application using the RTA-OSEK GUI. You can use the concepts explained in this guide to create your first RTA-OSEK application.

You may find that in this chapter you see things that you haven't learnt about yet. If this happens you can use the other chapters of this guide to find out the information that you need.

In RTA-OSEK v4, the GUI has been split into three views – these are accessed by using the tabs at the lower-left of the GUI (see Figure 3:1):

- The *Planner* is described in Sections 3.1 to 3.4, and will be familiar to users of previous versions of RTA-OSEK. This is the preferred way of describing an application since verification of the application's design can be carried out at this stage.
- The *Builder* is for developers who are familiar with OSEK concepts and simply want to construct a system without using the design analysis/verification aspects of the *Planner*. This is described in section 3.5.
- Finally, the *RTA-Trace* view allows configuration of RTA-OSEK parameters related to the LiveDevices RTA-TRACE product (a software logic-analyzer for embedded systems – contact your local sales office for further details).

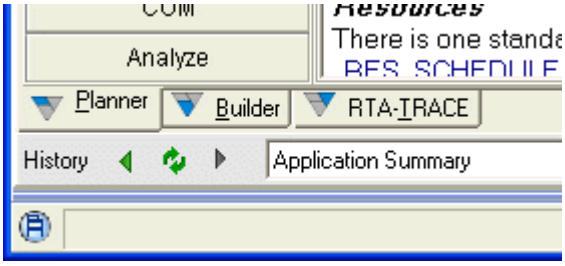


Figure 3:1The view tabs

3.1 Overview

The process of creating a new application in the RTA-OSEK *Planner* has a number of stages. The diagram in Figure 3:2 shows how the development lifecycle works and how the steps fit together.

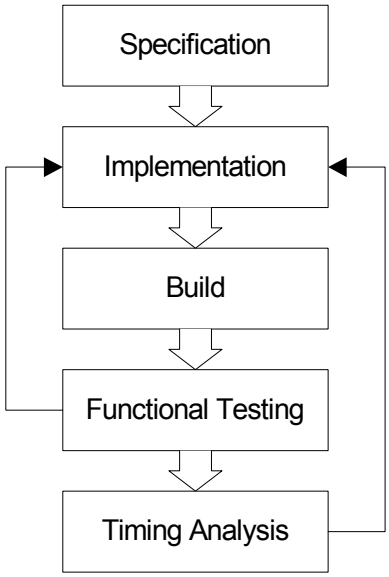


Figure 3:2 - The Development Process

Figure 3:2 shows that a **specification** should exist before creating a new application. The application can then be constructed and the **implementation** can begin using the supplied specification. Once coding is complete you must **build** the application, before starting the **functional testing** and, optionally, the **timing analysis**.

At the functional testing and timing analysis stages, the implementation may change, so the application must be built and tested again until it is finished.

Each of these steps is explained in detail in the following sections.

3.1.1 Specification

When a new target application is being developed, a specification should be supplied. You can see an example specification in Section 3.3.1.

A specification tells you things like:

- The target platform details.
  These details include the processor type, clock speed and available memory.
- A list of external real-world inputs to the system.
  The real-world inputs are called **stimuli**. Stimuli are things like switches being closed, timers expiring, network messages being received and certain angular positions being reached in an engine. You can also think of time as a real-world input. For example, if you have to poll hardware, you might create a stimulus that occurs every 10ms.
- A list of outputs from the system.
  The outputs are called **responses**. Responses describe how the system responds to stimuli. Responses might include, for instance, turning a lamp on, updating an internal count, activating a motor or sending a message to another controller.

- Performance requirements.
  For each stimulus, there will be at least one response that the system has to make. This response will have to be made within a time limit. The **deadline** is the latest time that the response is allowed to occur after the stimulus. Specification of system deadlines is important for timing analysis. The purpose of timing analysis is to show whether or not the system's requirements will be met in the worst case system loading.

Let's look at a real-world example. In Figure 3:3, you can see a car hitting an object during an impact test – when the car hits the object, the airbag inflates. The car hitting the object is the stimulus; the airbag inflating is the response to the impact. For the response to be effective, it must occur before the deadline. In this case, the deadline for the airbag inflation must be set to minimize the chance of injury to the vehicle's occupants.

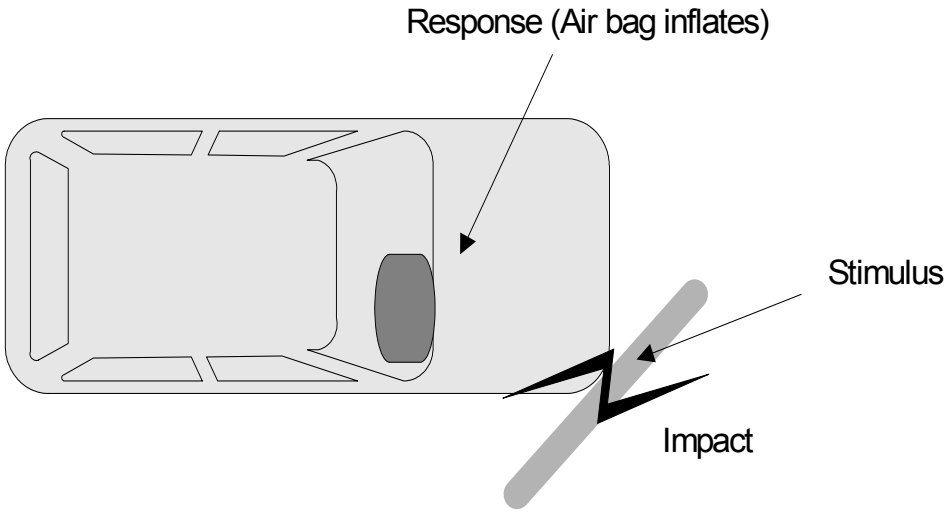


Figure 3:3 - Responding to a Stimulus

3.1.2 Implementation

Once the specification stage is complete, you'll then need to think about the implementation. In the implementation phase, you will decide how the target detects the specified stimuli and how the responses are implemented.

External stimuli are often detected by raising hardware interrupts. An interrupt service routine (ISR) will run when the target processor responds to the interrupt.

Usually the ISR will activate a specific task that implements a response, although an ISR can implement a response directly, if required. Having short ISRs and appropriately prioritized tasks will give the most responsive results, particularly in heavily loaded systems.

On some targets each interrupt source has its own entry in the processor vector table, so one ISR is needed for each interrupt that can occur. Other targets allow several interrupt sources to use the same vector, so an ISR may need to decode which interrupt source is active.

In simple systems, each response can be implemented in a separate task. The responses with the shortest deadlines should be assigned to the tasks with the highest priority. This gives the task the best chance of meeting the deadline. You can use *schedulability analysis* to confirm whether or not the task will always meet the deadline.

A task can implement more than one response. For example, in Figure 3:4 you can see that stimulus `S` can be specified to result in response `R1` within time `T1`, then response `R2` within time `T2` and then response `R3` within time `T3`.

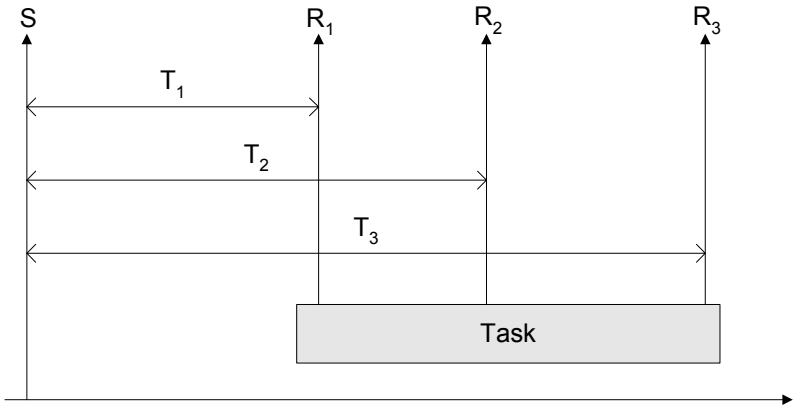


Figure 3:4 - A Task Implementing More Than One Response

You can usually use a single task to implement all of the responses in Figure 3:4. RTA-OSEK will be able to confirm whether or not it can meet each deadline.

If the task can determine which stimulus it is responding to, it is also possible for a task to provide responses for more than one stimulus. There are a number of mechanisms that can be used to pass information to a task on which stimulus has occurred:

- Global variables.
- OSEK COM messages.
- OSEK OS events.

Once the structure of the application, in terms of tasks and ISRs, has been established it is then refined using OS features including resources, queuing mechanisms, events and messages.

RTA-OSEK can provide skeleton source code files for each task and ISR that you need to implement. It is up to you to write the code that executes when the tasks and ISRs run.

The RTA-OSEK *Planner* provides an implementation tick-list for your application showing the code that needs to be implemented. Once coding is complete the various files can be compiled and linked to generate the application executable file. You will find out how to implement example applications in Section 3.3.2.

Each of these pieces of code (implementations of Tasks and ISRs) should be written in its own C source file. There are a number of software engineering issues associated with this:

- Task threads are isolated in the application source code. This is good development practice and allows your compiler to provide some protection against certain classes of bug. Using static variable declarations in a file, for example, protects those variables against being accidentally changed by other tasks or ISRs.
- Configuration management is made easier because changes to tasks are limited to a single task in a single file.
- Testing is made easier because it can be performed on a per task basis. Individual tasks can be replaced with stubs, where necessary, for integration with automated test management tools.

When using RTA-OSEK, it is strongly recommended that you put the code relating to each task and ISR into individual files. This is because RTA-OSEK automatically creates task and ISR specific header files that can be included with the C source code. These header files contain the static interface for each task or ISR.

Static interface header files allow optimized versions of the operating system API calls to be selected at compile time, based on RTA-OSEK's knowledge of the application. This approach allows you to take advantage of some significant code optimizations.

If you choose not to use the static interface, then all source files that use the RTA-OSEK component API must `#include` the file `osek.h` (or `oseklib.h` if you are creating code to go in a library), rather than the task or ISR-specific files. Your application will have the same functionality, but will be slower and larger.

If you must have multiple tasks or ISRs in a single source file, you must not mix tasks that use heavyweight and lightweight termination. You must `#define OS_HEAVYWEIGHT` or `OS_LIGHTWEIGHT` as appropriate and only `#include "osek.h"`.

3.1.3 Build

When the basic structure of your application is complete, the RTA-OSEK GUI can 'build' the assembler, C and header files that are needed to assemble/compile and link with your own source code files. You can then create your executable application. You can find out more about building an application in Section 3.5.2.

3.1.4 Functional Testing

For functional testing the application must be downloaded to the target hardware. The first time you test an application it is recommend that, if possible, only one stimulus be triggered at a time.

If you discover any problems during these tests you should modify the application, rebuild it and then retest it. You should repeat these steps until

you are confident of the functionality of the application. The functional testing stage is explained in Section 3.3.4.

3.1.5 Timing Analysis

The final stage of testing is to prove that the application will meet its timing deadlines. To do this, you will need to measure the execution times of your code and enter this information into the RTA-OSEK GUI.

Measurement of execution times can be complex, but this information is required to obtain accurate timing analysis.

RTA-OSEK tells you whether your system is **schedulable** or not. An application is schedulable when all deadlines will be met. RTA-OSEK can tell you which deadlines are not met for a system which is not schedulable.

You can try to make the system schedulable by:

- Indicating the maximum allowed execution time for each task or ISR.
- Rearrangement of task priorities.
- Adjustment of the CPU clock.

You can find out more about analysis in Section 3.3.5.

3.2 A Simple Example

In this section you will see how to build a simple application by configuring OSEK OS objects directly from the RTA-OSEK GUI.

Our system specification is as follows:

- The target processor is the Motorola HC12. (Select a different target if your installation does not include this processor.)
- The target clock is 8MHz.
- Incoming CAN bus messages will generate an interrupt to the CPU. An ISR must handle the initial processing of each message and then activate a worker task to complete the processing at a later time.
- Three tasks must run periodically at 3ms, 6ms and 14ms rates.
- The 14ms periodic task shares a data buffer with the CAN worker task. Mutually exclusive access to the data buffer must be enforced to avoid data corruption.

3.2.1 Creating a New Application using the RTA-OSEK GUI

To create a new application, you'll need to run the RTA-OSEK GUI and select **New** from the **File** menu. The **Select Target** dialog will open. Select the target processor from the **Available Targets** and **Variant** drop down lists.

In Figure 3:5, the *HC12/COSMIC 16 task* target has been selected and the *HC12* variant is being used. Remember that if your installation does not include the HC12, you must select a different processor.

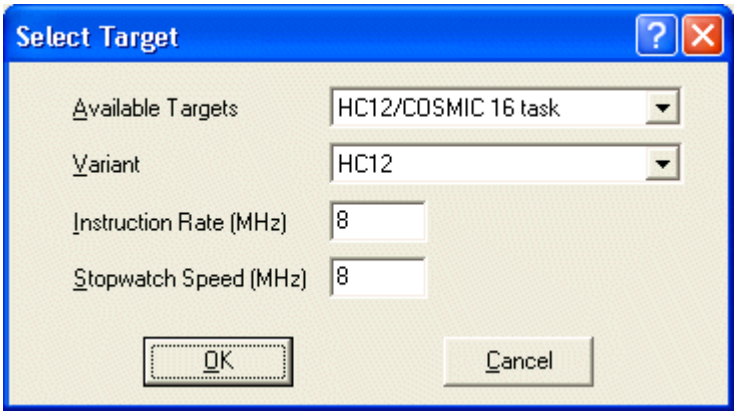


Figure 3:5 - Selecting a Target Processor

Important: The **Available Targets** list will only show the targets that have been installed on your own computer, according to your license file. Please contact LiveDevices if you cannot see the targets that you expected to. Each target may have a number of variants to reflect different chip versions based on a common processor core.

The **Select Target** dialog in Figure 3:5 can also be used to enter the **Instruction Rate** and **Stopwatch Speed**. The instruction rate should be set according to the smallest instruction cycle in the processor.

The stopwatch speed value depends on the sample rate used by timer hardware attached to the `GetStopwatch()` function that is used in Timing and Extended builds. This function is used to measure execution time in OS and application code. Ideally the stopwatch is run at the instruction rate, but on some targets this may not be possible.

When the target information has been set and the **OK** button has been clicked, the RTA-OSEK GUI automatically displays a summary of the new application. You can see this in Figure 3:6. You can refer back to this summary at any stage to see an overview of the entire system that you are creating.

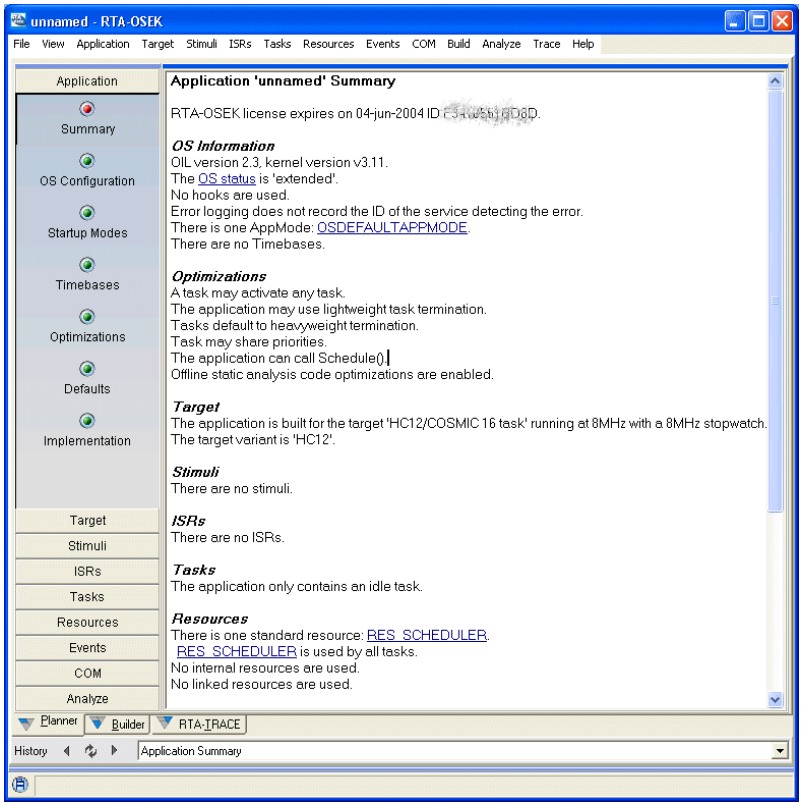


Figure 3:6 - Viewing a Summary of the New Application

3.2.2 Saving the Application

As soon as you create a new application, it is a good idea to save it. To save an application for the first time, from the **File** menu select **Save As...**.

In the **Save As** dialog, use the **Save In** list to navigate to the location that you want to save the file in. For the **File Name** in this example, enter the name **UserApp**. Click the **Save** button to save your new application.

You can save the application at any time by using the **File** menu to select **Save** or by pressing the **Ctrl** key and the **S** key together on the keyboard (**Ctrl+S**).

3.2.3 Viewing the OIL File

The file that is created is saved using OIL v2.3 syntax. This means that other OSEK compatible tools can read it. You can view the contents of the OIL file for the current application by clicking on the **View** menu and selecting **OIL File**.

You'll see the OIL file contents displayed in the lower half of the workspace, so it will look something like the example in Figure 3:7.

In the upper part of the window you can see the details that were displayed in the workspace. In the lower part of the window you can now see the OIL file.

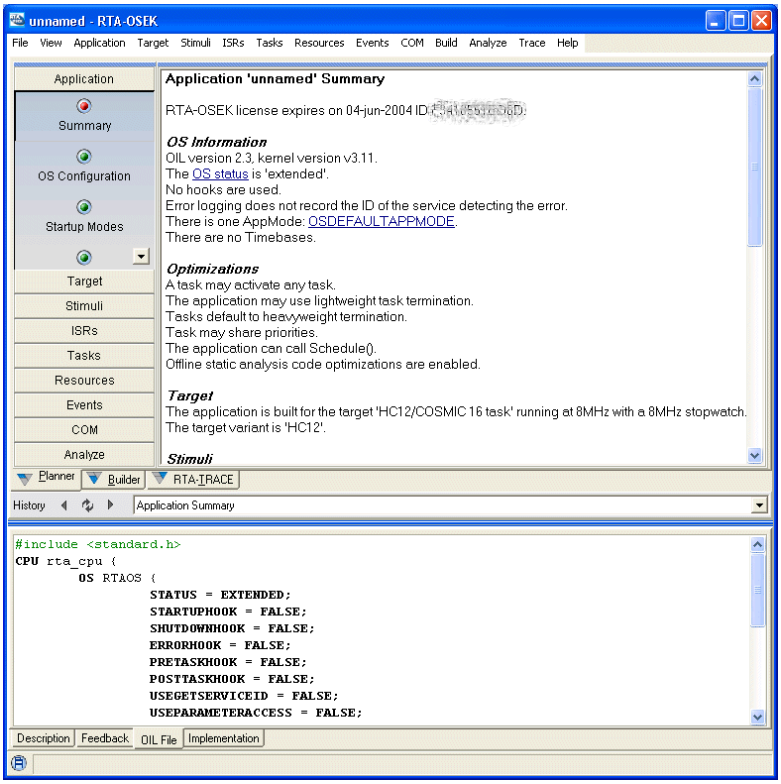


Figure 3:7 - Viewing an OIL File

You can close the OIL view by clicking on the **View** menu and deselecting **OIL File**.

3.2.4 Implementation

You have now reached the implementation stage where you will learn how to configure the RTA-OSEK OS objects, such as tasks, ISRs, counters and alarms, which make up the application.

In many of the following steps, you will be required to carry out certain actions on instances of OS objects; these actions are accessed from a common icon set, shown in Figure 3:8 – from the left, you can see the **Add**, **Rename**, and **Delete** icons.



Figure 3:8 - Common Icons (Add, Rename, Delete)

Creating ISRs

This example has two interrupt sources - one that detects the arrival of a CAN message and another that is attached to a hardware timer that can provide interrupts every 1ms.

To create a new ISR, select the **ISRs** group from the navigation bar.

Figure 3:9 shows how the **ISRs** group is selected and how the ISR Summary initially appears in the workspace.

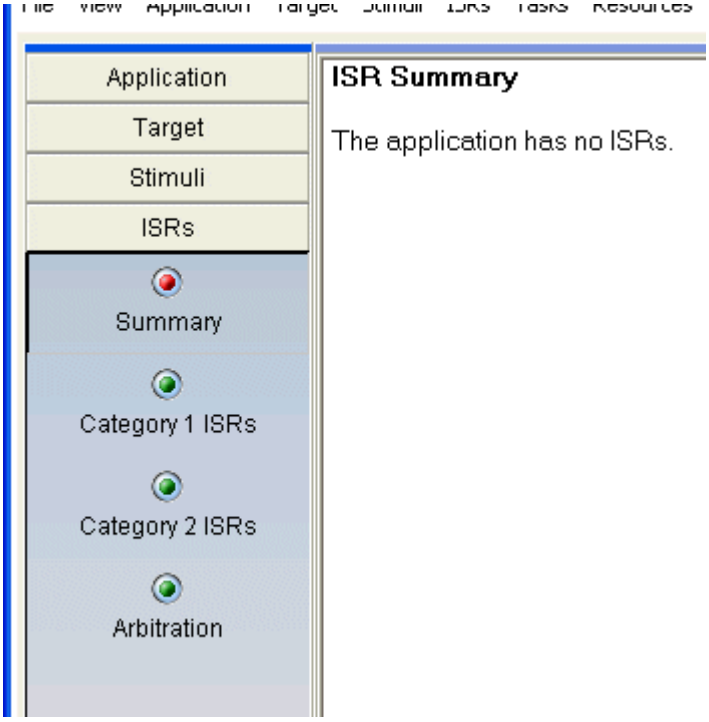


Figure 3:9 - Interrupts

The ISRs to be added will be Category 2 ISRs since OS calls are going to be made from them (Category 1 ISRs are forbidden from making OS API calls).

The first ISR to be added will be responsible for handling a 1ms tick generated by a hardware timer.

- From the navigation bar, select the **Category 2 ISRs** subgroup.
- Create the ISR by clicking the **Add** button in the workspace.
- In the Add Cat 2 ISR dialog, enter the name ***TimerISR*** and click **OK.**
- Depending upon the target type, you may need to enter an interrupt **Vector** and a **Priority**. Here we attach it to a timer peripheral – see Figure 3:10.

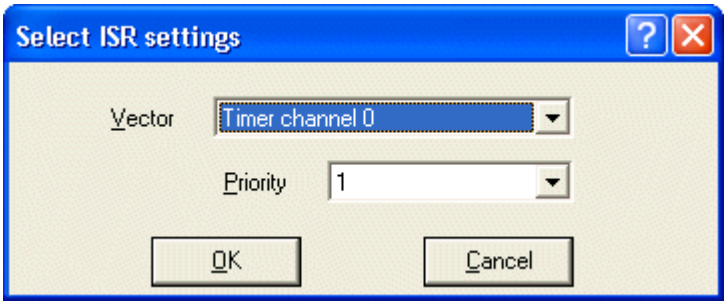


Figure 3:10 - Setting Priority and Vector for an ISR

The workspace displays the default settings for the new ISR – see Figure 3:11.

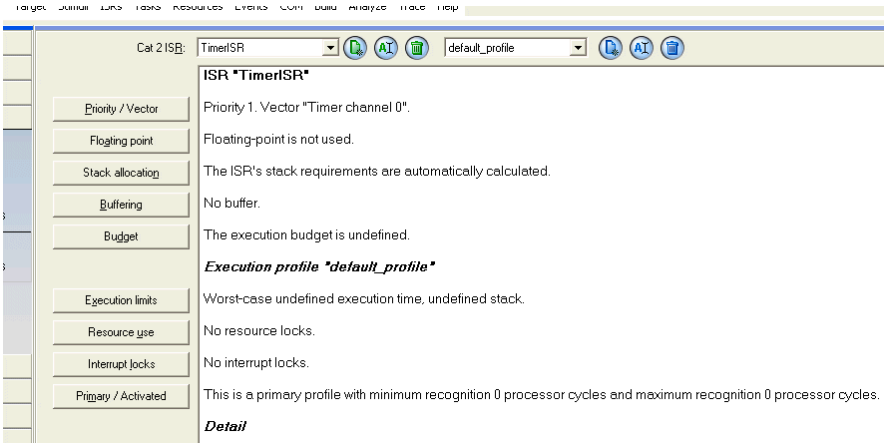


Figure 3:11 - Properties of the Newly-Created Interrupt

Now repeat this procedure to create a Category 2 ISR called *CanISR*, responsible for handling the interrupts generated by incoming CAN messages.

Creating Tasks

You will now create the four tasks that perform the work of the application. Remember that three of these tasks are activated periodically at rates of 3ms, 6ms and 14ms. The fourth task is activated by *CanISR*.

To create a new Task, select the **Tasks** group from the *Planner* navigation bar.

Figure 3:12 shows how the **Tasks** group is selected and how the **Task Summary** initially appears in the workspace.

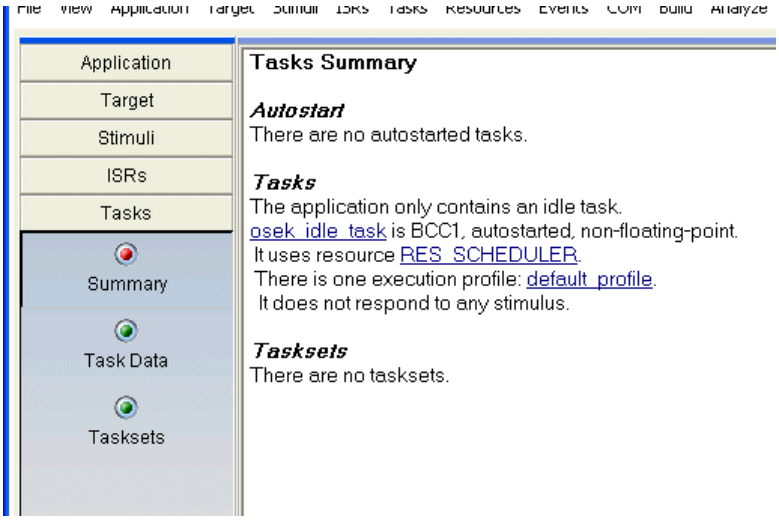


Figure 3:12 - System with no Tasks

- From the navigation bar, select the **Task Data** subgroup.
- Create a task by clicking the **Add** button in the workspace.
- In the **Add Task** dialog, enter the name ***Task1*** and click **OK.**

- In the **Task "Task1" priority** dialog (Figure 3:13), enter a priority of ***10*** for this task.

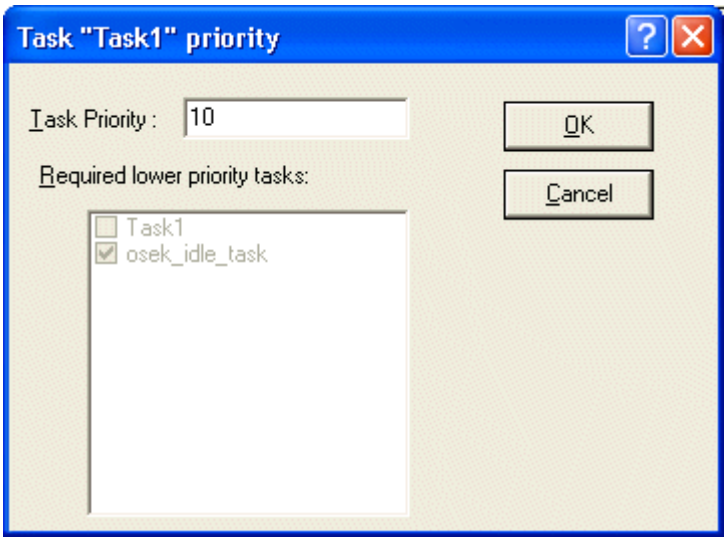


Figure 3:13 - Setting Task Priority

The workspace displays the default settings for the new task, as shown in Figure 3:14

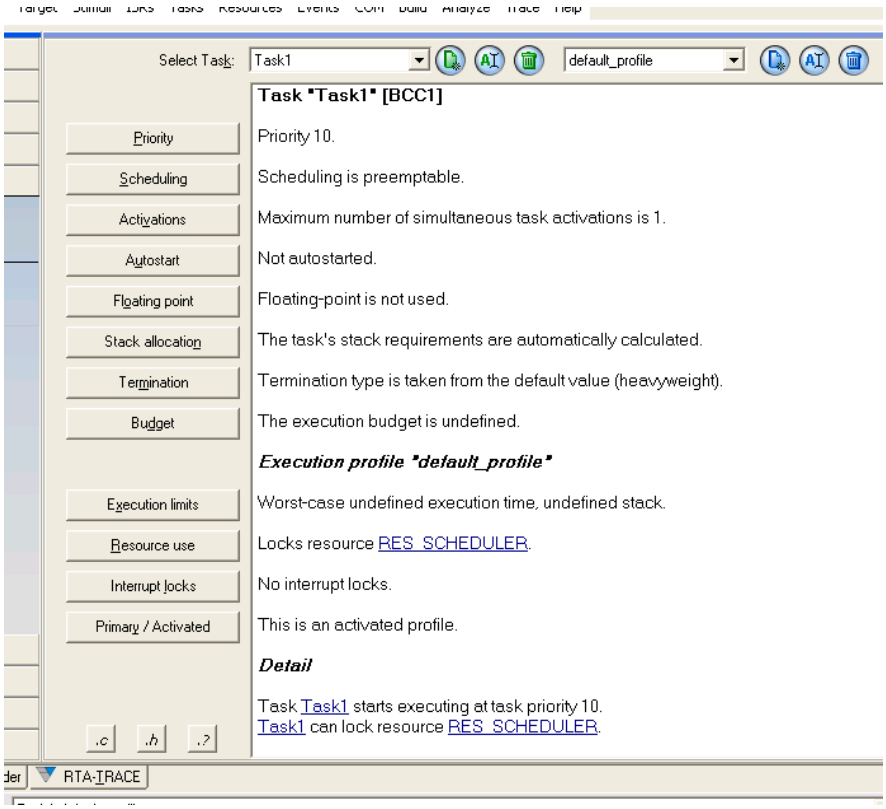


Figure 3:14 - Task Properties

Now repeat this procedure to create another three tasks as follows:

- *Task2* with priority 9
- *Task3* with priority 8
- *CanWorker* with priority 3

Creating a Counter and Alarms

The three periodic tasks that have been created will be activated by the expiry of alarms attached to a counter. For this example, we will 'tick' the counter from *TimerISR* (invoked every 1ms), thus a counter tick is equivalent to 1ms. Alarms attached to the counter will expire after a specified number of ticks. When each alarm expires, an associated task will be activated by the OS.

To create a new counter, select the **Stimuli** group from the navigation bar.

Figure 3:15shows how the **Stimuli** group is selected and how the **Stimuli Summary** initially appears in the workspace.

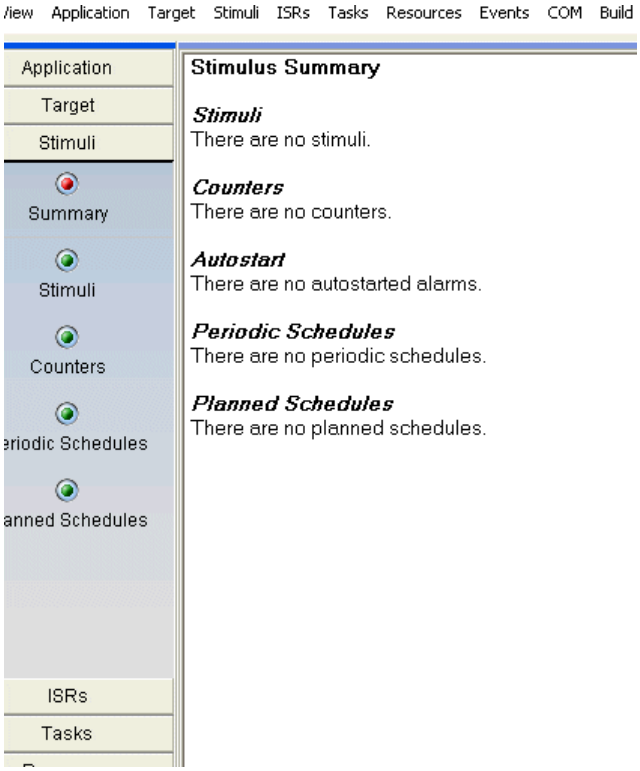


Figure 3:15 - System with no Alarms

- From the navigation bar, select the **Counters** subgroup.
- Create a counter by clicking the **Add** button in the workspace.
- In the **Add Counter** dialog, enter the name ***TimerCounter*** and click **OK.**
- In the **Tick rate for timebase "TimerCounter"** dialog (Figure 3:16), enter a fastest tick rate of 1 realtime ms.

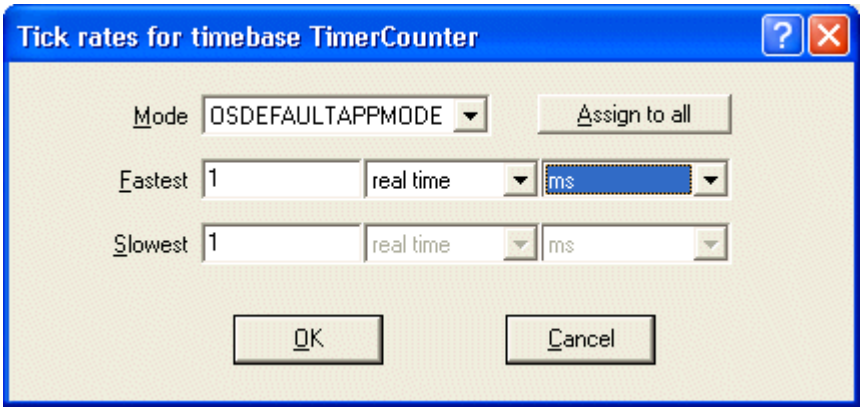


Figure 3:16 - Setting Timer Tick Rate

The workspace displays the default settings for the new counter (Figure 3:17).

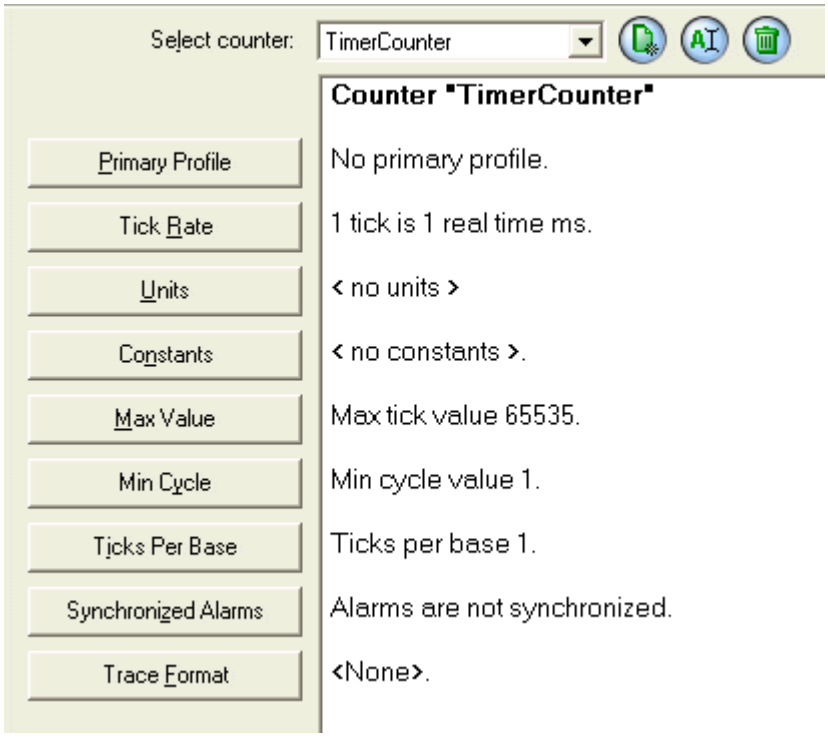


Figure 3:17 - Counter Properties

Finally we need to indicate that the counter is ticked by *TimerISR*:

- Click the **Primary Profile** button and select ***TimerISR*** from the primary profiles dropdown list and click **OK**.

Now we need to create the alarms responsible for activating the three tasks created previously. *Task1* is to be run every 3ms, *Task2* runs every 6ms, and *Task3* runs every 14ms.

- From the navigation bar, select the **Stimuli** subgroup.
- Create a stimulus by clicking the **Add** button in the workspace.

- In the **Add stimulus** dialog, enter the name ***Task1Alarm*** and click **OK**.
- Click on the **Arrival Type** button and select **periodic**.
- Click on the **Schedule/Counter** button and select ***TimerCounter*** as the counter for this alarm.
- In the **Arrival Pattern** dialog, enter a cycle time of 3 ***TimerCounter*** ticks (equivalent to 3ms).

The workspace, as shown in Figure 3:18, displays settings for the new alarm.

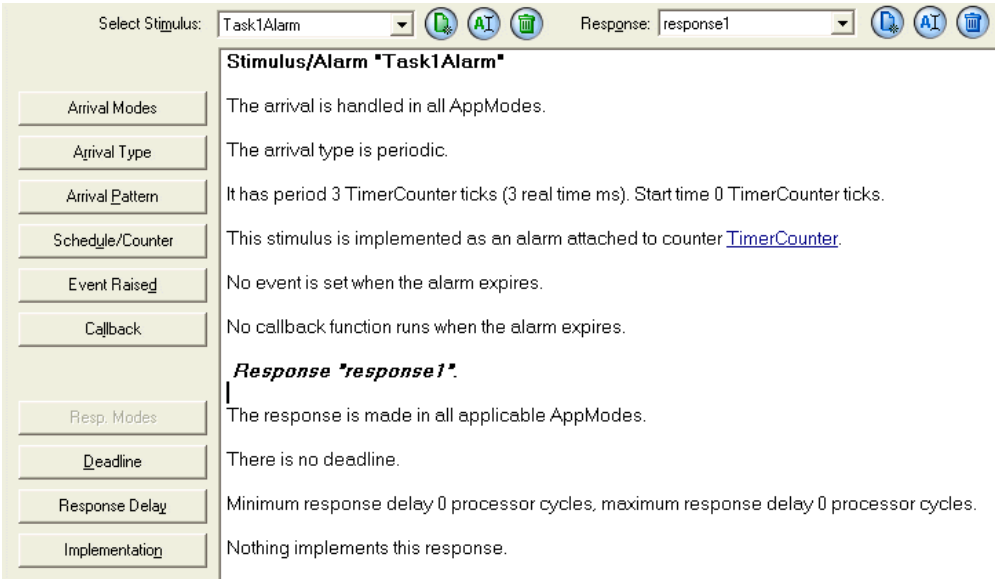

Select Stimulus:
Task1Alarm
Response:
response1
Stimulus/Alarm "Task1Alarm"
Arrival Modes
The arrival is handled in all AppModes.
Arrival Type
The arrival type is periodic.
Arrival Pattern
It has period 3 TimerCounter ticks (3 real time ms). Start time 0 TimerCounter ticks.
Schedule/Counter
This stimulus is implemented as an alarm attached to counter TimerCounter.
Event Raised
No event is set when the alarm expires.
Callback
No callback function runs when the alarm expires.
Response "response1".
Resp. Modes
The response is made in all applicable AppModes.
Deadline
There is no deadline.
Response Delay
Minimum response delay 0 processor cycles, maximum response delay 0 processor cycles.
Implementation
Nothing implements this response.

Figure 3:18 - Alarm Properties

We now need to make the alarm activate *Task1* - this is the **response** to the alarm **stimulus**.

- Click on the **Implementation** button, and select ***Task1*** in the **Implementer** drop-down – there is no need to enter anything in the **Execution time** field yet.

You can now repeat this procedure to create two other alarms as follows:

- *Task2Alarm* which activates *Task2* every 6ms;
- *Task3Alarm* which activates *Task3* every 14ms

Creating a Resource

Resources are used to enforce mutually exclusive access to a critical section of application code. This is usually to prevent corruption of data in a global variable. In this application you must create a resource that is shared between the *CanWorker* and *Task3* tasks. The task that has successfully locked this resource can safely modify a data buffer without the other task disrupting it.

To create a new resource, select the **Resources** group from the navigation bar.

Figure 3:19 shows how the **Resources** group is selected and how the Resource Summary initially appears in the workspace.

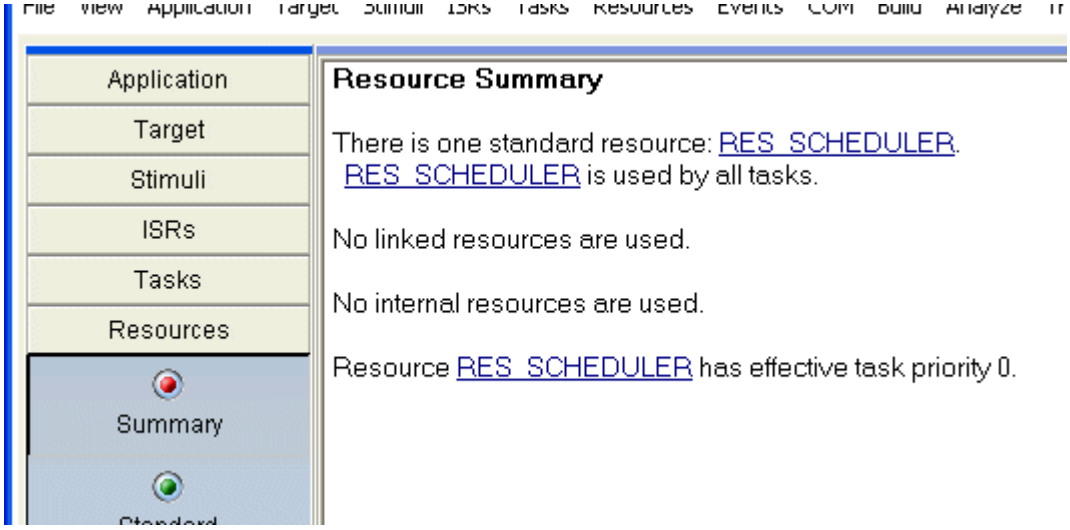


Figure 3:19 - System with no User-Declared Resources

- Select the **Standard** subgroup from the navigation bar.
- Create a resource by clicking the **Add** button in the workspace.
- In the **Add resource** dialog, enter the name ***CanResource*** and click **OK**. The workspace displays the default settings for the new resource.

Now that the resource has been created, we need to indicate which tasks use it:

- Click the **Change Users** button, select ***Task3*** and ***CanWorker*** in the **Select Users** dialog and click **OK**. RTA-OSEK automatically calculates the effective task priority of this resource.

The resultant workspace can be seen in Figure 3:20.



Figure 3:20 - Resource Properties after Users are Added

Writing Task and ISR Code

You will now need code for the ISRs *TimerISR* and *CanISR*, tasks *Task1, Task2, Task3* and *CanWorker*, as well as the application's `main()` function. The `main()` function includes the application startup as well as the OS idle mechanism.

The C source code can be created outside of the RTA-OSEK GUI if you wish, but you can also create templates from the RTA-OSEK GUI to help get you started. To do this:

- Change to the *Builder view*, then from the navigation bar, select the **Custom Build** subgroup.
- In the workspace, click the **Create Templates** button. RTA-OSEK will create seven C source files and put skeleton code in each of them. It also creates a batch file `rtkbuild.bat`, which you will use later in the build phase.

Writing Code for TimerISR and CanISR

Move back to the *Planner view* and select the **ISRs** group from the navigation bar. Then select the **Category 2 ISRs** subgroup and from the workspace, select ***TimerISR***.

You don't have to worry if you can't remember everything that has to be done in the ISR, because the RTA-OSEK GUI can tell you. Simply select the **Implementation** option from the **View** menu. The lower section of the ISR window is displayed and the implementation details for the ISR will appear. You can resize this window by moving your mouse over the blue horizontal 'splitter' bar. Click and hold the left mouse button and drag the bar up or down.

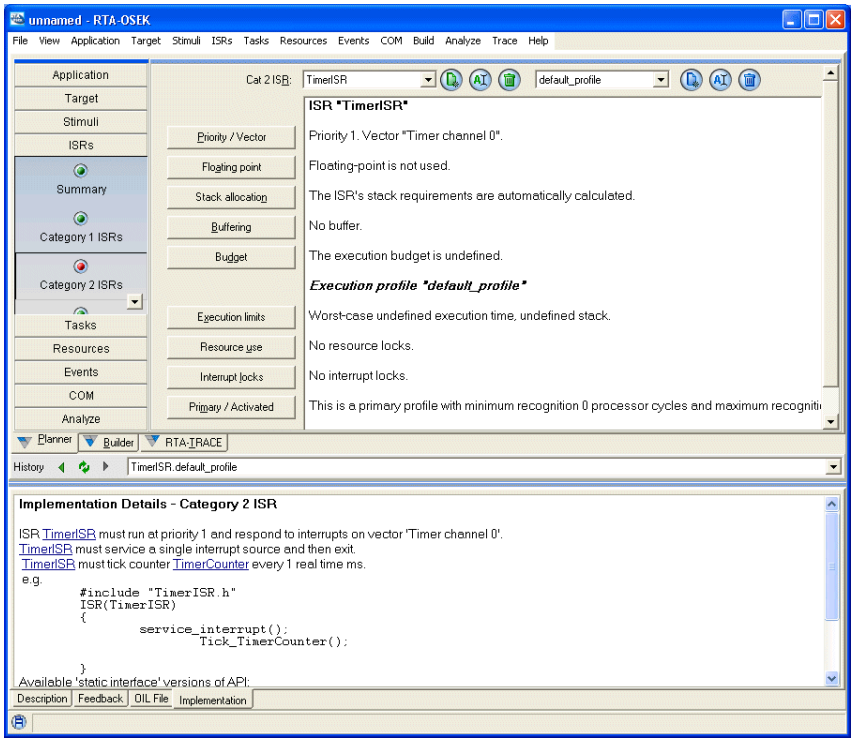


Figure 3:21 - Viewing the TimerISR Implementation Notes

The sample code shows the ISR-specific header file `TimerIsr.h` being `#included`, followed by the ISR body. Since we have made this ISR the primary profile for the counter *TimerCounter*, the implementation view indicates that this ISR is required to call `Tick_TimerCounter()`. This call 'ticks' the *TimerCounter* so that the counter is made aware of time passing. You must ensure that your timer hardware is configured so as to cause these 'ticks' to happen at the defined rate (1ms in this example). Otherwise, the alarms attached to this counter will not expire at the correct times and the various tasks will not be activated as desired.

You can close the implementation notes by deselecting the **Implementation** option in the **View** menu.

You can directly edit the source code for the ISR by selecting the  button in the Category 2 ISRs workspace.

The code for *CanISR* can be written in a similar way. In this case, you should add an `ActivateTask_CanWorker()` call to activate the *CanWorker* task in the ISR body.

Important: When editing files from within the RTA-OSEK GUI, the default editor is set to be the Windows Notepad application. You can select your own preferred editor from the **File** menu, by selecting **Options**.

Portability Note: The code that needs to be written here is target-specific, so no details are given where lines involve detection of pending interrupt sources and how they are acknowledged.

Writing Code for Task1

From the **Tasks** group on the navigation bar, select the **Task Data** subgroup. Then select the task ***Task1***.

Use the **Implementation View** to check the code that is required for this task.

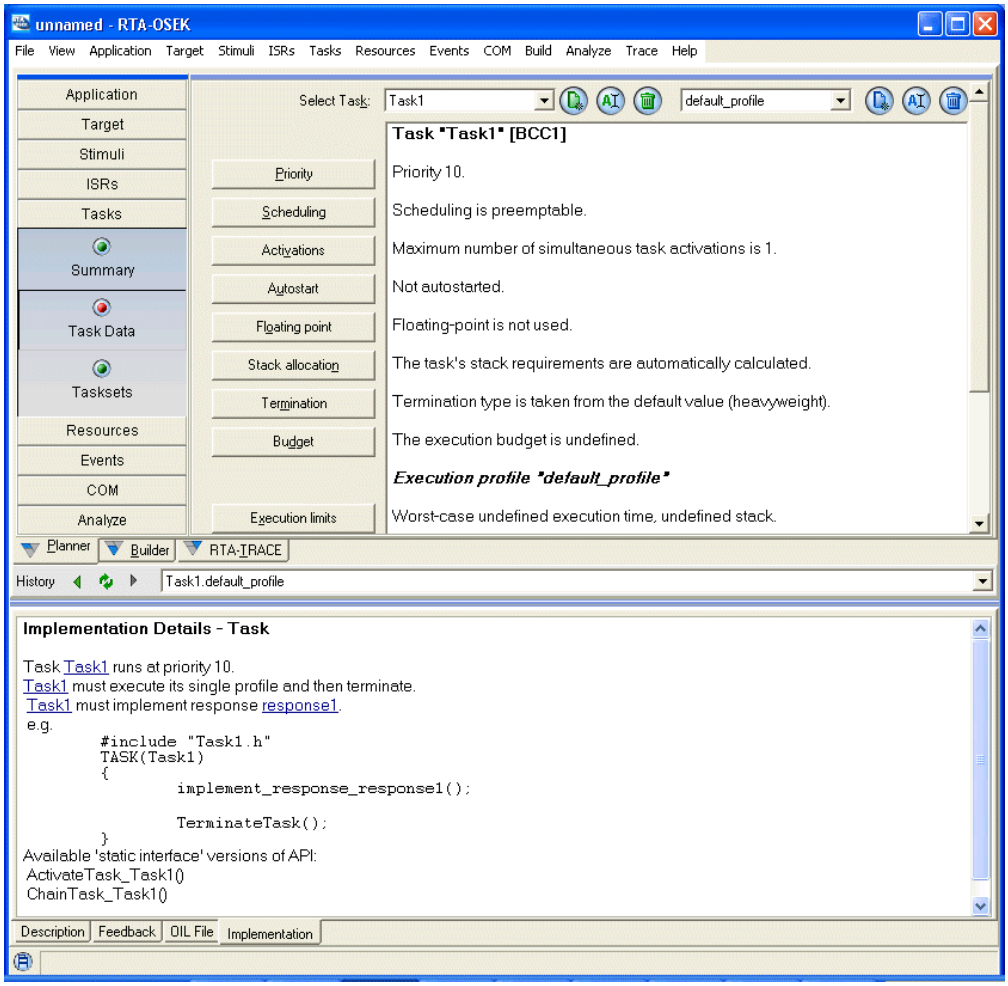


Figure 3:22 - Viewing the Implementation Notes for Task1

As with the ISR view, you can directly edit the source code for the task by selecting the .c button in the workspace. The code you write will be target-specific, but should follow the structure in the implementation view.

Writing Code for the Remaining Tasks

The code for tasks *Task2*, *Task3* and *CanWorker* follow the same pattern.

Whenever you modify the RTA-OSEK configuration always check that the suggested implementation matches the code you have written.

Writing Code for 'main'

In C programs, `main()` is the starting point for the main application. It is called after the low-level startup initialization. Usually interrupts are disabled prior to `main()` being entered.

The skeleton code generated by the RTA-OSEK for `main()` is shown in Code Example 3:1.

```
/* Template code for 'main' in project: UserApp */

#include "osekmain.h"

OS_MAIN()
{
  StartOS(OSDEFAULTAPPMODE);
  ShutdownOS(E_OK);
}
```

Code Example 3:1 - Template Code for main()

Note that the `OS_MAIN()` macro is used rather than `main()`. Individual compilers have different criteria for the arguments and return types that are allowed for `main()`, so RTA-OSEK provides `OS_MAIN()` to assist portability.

Portability: Using `OS_MAIN()`, rather than `main()`, in applications can help make them more portable to different RTA-OSEK targets.

The `StartOS(OSDEFAULTAPPMODE)` call is used to start the operating system. No operating system API calls should be made before `StartOS()` is called.

The `ShutdownOS()` call is used to stop the OS when (and if) the application completes. The default action for `ShutdownOS()` is to stay in an infinite loop and not to return. This call is not normally used because applications tend to run 'forever' (or until the processor loses power or is reset).

In your example application, you need to perform some initialization of the target hardware before calling `StartOS()`. This makes sure that the timer is set to interrupt every 100ms and the appropriate interrupt sources are enabled.

After `StartOS()`, you must set up the alarms that are used by the application. Use the `SetAbsAlarm()` API call to do this. `SetAbsAlarm()` takes three parameters: the name of the alarm that is being set up, its start time and its cycle time. The start time is the first time that the alarm will expire at. Be careful if you set this to 0. This will mean that the alarm counter must cycle through its entire range before wrapping around to 0. This can take a long time on some hardware. The cycle time sets up the periodic expiry of the alarm after the start time. In this application, each alarm is set to start at 1ms.

The cycle times for *Task1Alarm*, *Task2Alarm* and *Task3Alarm* are 3ms, 6ms and 14ms respectively.

The code that executes after `StartOS()` belongs to the idle task. The idle task is called `osek_idle_task`. The idle task can act like any other task; it can make API calls, use resources, send and receive messages, send and wait for events and so on. It cannot be directly activated because it only terminates when `ShutdownOS()` is called and it cannot use internal resources because it would prevent other tasks from starting.

Important: Putting code in the idle task can be a very efficient way of implementing a system. In particular, if you have only one task that needs to respond to OSEK events you should use the `osek_idle_task`. Your application will be significantly smaller and more responsive if the idle task waits for events, rather than any other task.

The RTA-OSEK GUI will show you a suggested implementation for `OS_MAIN()`. Select the ***osek_idle_task*** task in the **Task Data** subgroup (in the **Tasks** group on the navigation bar) and view the implementation details.

Note that in this case, the idle task does no work. On targets that support it, you can put the processor into a 'sleep' state in the idle task. The processor must 'wake-up' if an interrupt occurs.

Important: The idle task must not terminate. It must loop forever.

Setting up Timer/Counter Hardware

In Code Example 3:2, the function `do_target_initialization()` needs to initialize the interrupt sources to drive *TimerISR*. One of these sources is a hardware counter/timer that needs to provide an interrupt every 1ms.

You may wish to use code based on Code Example 3:2 to do this.

```
void do_target_initialization(void)
{

  unsigned int timer_divide;
  timer_divide = OSTICKDURATION_TimerCounter /
OS_NS_PER_CYCLE;

  /* Initialize the timer hardware */
  SetupTimer(timer_divide);

  /* Other target initialization */
  ...

}
```

Code Example 3:2 - Initializing Timer Hardware

Code Example 3:2 shows initialization using the two RTA-OSEK-generated constants `OSTICKDURATION_TimerCounter` and `OS_NS_PER_CYCLE`.

The `OSTICKDURATION_TimerCounter` constant specifies the duration of the 'tick' of the counter in nanoseconds (ns), so in this example the `OSTICKDURATION` of 1ms is 1,000,000ns.

The `OS_NS_PER_CYCLE` constant specifies the duration of the CPU instruction cycle in ns. For an 8MHz CPU, this is 125ns.

In this example, you require the timer to be configured to interrupt every 1ms. If you use these constants to calculate the divide ratio, the code will automatically adjust if the clock rate changes.

OS Status, ErrorHook and Callbacks

For preliminary testing, you should run the application using the operating system's **Extended** build. Extended build means that the OS performs rigorous checks in each API call. Of course, this takes time and code space.

Once the application is seen to be working correctly, you will usually switch to **Standard** build. Very few checks are made with Standard build, so the OS can run much more efficiently.

To choose Extended or Standard status:

- Select the **Application** group from the *Planner* navigation bar and, select the **OS configuration** subgroup.
- Click on OS Status, and choose the appropriate status level.

When using Extended build, you can check the return status code from each API call or alternatively request that Error Hook be used. This is a function that the OS will call whenever an error is detected. You write the implementation of Error Hook in your application. Normally you will use it to halt debugging and to alert you of errors.

To use the Error Hook facility

- Select the **Application** group from the *Planner* navigation bar and then select the **OS Configuration** subgroup.
- Click the **Hooks** button. The **Select Hooks** dialog opens.
- Select the **Error Hook** checkbox and then click the **OK** button.

You can see that the Error Hook has been selected in Figure 3:23.

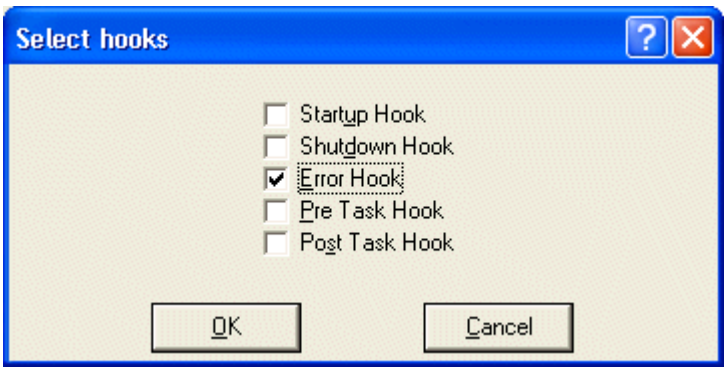


Figure 3:23 - Selecting the Error Hook

You can add the code needed to implement `ErrorHook()` in any source file, but `main.c` is a good place to start. Add the following code:

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType e)
{
 /* Put a debugger breakpoint here. */
 while (1) {
       /* Freeze. */
 }
}
#endif /* OSEK_ERRORHOOK */
```

Code Example 3:3 - The ErrorHook()

You can find more information about using `ErrorHook()` for debugging purposes in Section 13 of this User Guide.

There are three other functions that you must supply when using the Timing or Extended build. The operating system uses these to time the execution of your code.

Don't worry about the details at the moment; simply add the code in Code Example 3:9 after the `ErrorHook()`.

```
#ifdef OS_ET_MEASURE
OS_HOOK(void) OverrunHook(void)
{
  /* Put a debugger breakpoint here. */
  while (1) {
    /* Freeze. */
  }
}
OS_NONREENTRANT(StopwatchTickType)
GetStopwatch(void)
{
  /* Temporary implementation. A correct solution
   * returns the current stopwatch value. */
  return 0;
}
OS_NONREENTRANT(StopwatchTickType)
GetStopwatchUncertainty(void)
{
  /* Temporary implementation. A correct solution
   * returns the uncertainty in the stopwatch
   * value. */
  return 0;
}
#endif /* OS_ET_MEASURE */
```

Code Example 3:4 - Timing Callbacks

Final Checks

To view a complete implementation summary select the **Implementation** subgroup from the **Application** group on the navigation bar.

Use this as a checklist to ensure that your application is fully implemented. You can print out this summary by selecting **Print Selection** from the **File** menu.

3.2.5 Build

If you have successfully completed all of the steps in creating this example application, you can now start the build process. Switch to the *Builder*, and refer to Section 3.5.2 where the build process is described.

3.2.6 Functional Testing

The executable file can be downloaded to your target hardware, so that you can test its behavior.

Initial testing should always be performed using the Extended build with the Error Hook, because the OS will detect any misuse of API calls. Only once an application performs correctly should you switch to the Timing or Standard builds.

3.3 A Simple Example Using Timing Analysis

In this section you will see how to build a simple application using a stimulus-response model to capture the performance requirements and perform timing analysis on the result.

3.3.1 Your Specification

For this example, the specification contains the following requirements.

- The target processor is the Motorola HC12. (Select a different target if your installation does not include this processor.)
- The target clock is 8MHz.
- A button *Button1* can be pressed many times, but never faster than twice per second. The minimum interval is 0.1s.

Figure 3:24 illustrates these requirements (B is used in the diagram to indicate when the button is pressed).

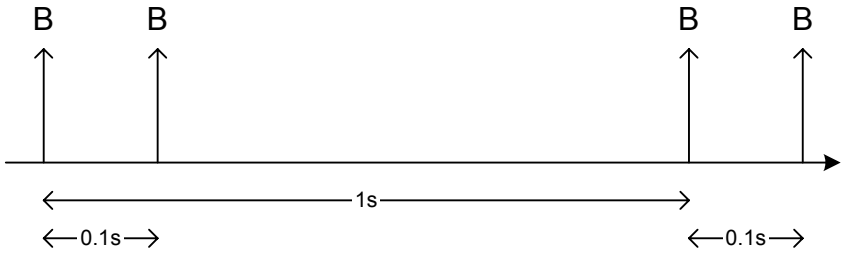


Figure 3:24 - Button1 Requirements

- A lamp *Lamp1* must be lit within 10ms of *Button1* being pressed.
- *Lamp2* must be switched off within 11ms of *Button1* being pressed.
- Motor *Motor1* must be started within 200ms of *Button1* being pressed.

Figure 3:25 shows these requirements.

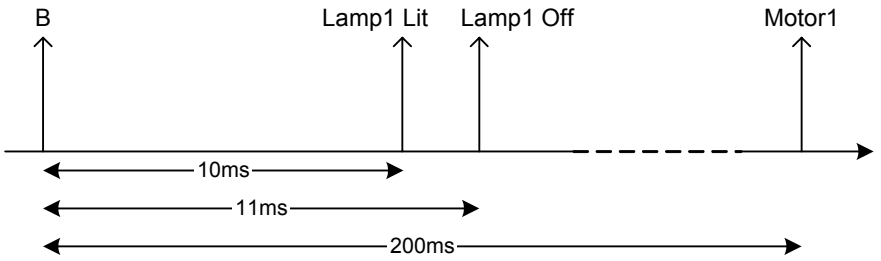


Figure 3:25 - Lamp1, Motor1 and Button1 Requirements

- When *Motor1* is up to speed, an interrupt is raised.
- *Lamp1* must be switched off within 11ms of the motor being up to speed.
- *Lamp2* must be switched on within 10ms of the motor being up to speed.

Figure 3:26 illustrates these requirements.

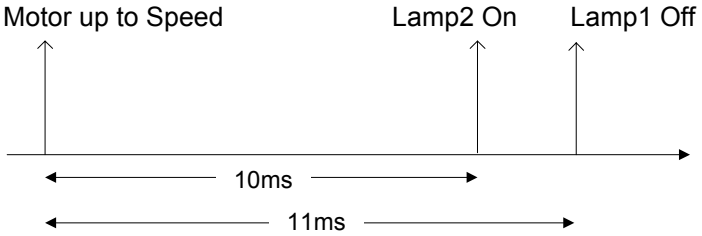


Figure 3:26 - Lamp1, Lamp2 and Motor1 Requirements

- *Lamp3* must toggle on/off every 1s, with an accuracy of +/- 2ms.

You can see this requirement in Figure 3:27.

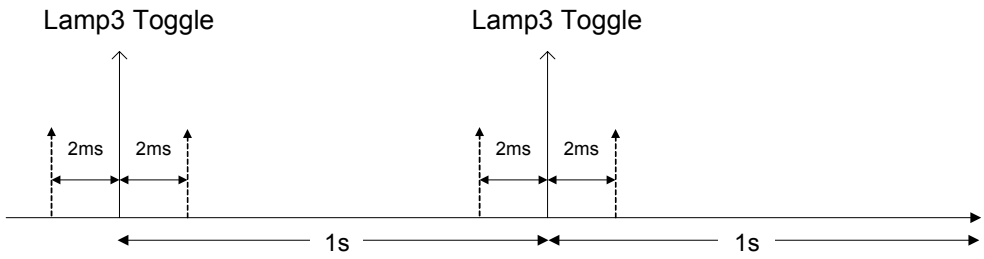


Figure 3:27 - Lamp3 Requirements

- The debounce circuitry attached to button *Button1* means that it takes between 0.1ms and 0.4ms from pressing the button to it being presented to the processor.
- It takes 0.5ms from the processor applying current to a lamp to the filament in the lamp being actually to be 'lit'.
- It takes 0.3ms from the processor removing current to a lamp to the filament in the lamp being deemed to be 'off'.
- It takes 50ms to start *Motor1* from the processor applying power to it.

Creating a New Application using the RTA-OSEK GUI

To create a new application, you'll need to run the RTA-OSEK GUI and select **New** from the **File** menu. The **Select Target** dialog will open. Select the target processor from the **Available Targets** and **Variant** drop down lists.

In Figure 3:28, the ***HC12/COSMIC 16 task*** target has been selected and the ***HC12*** variant is being used. Remember that if your installation does not include the HC12, you must select a different processor.

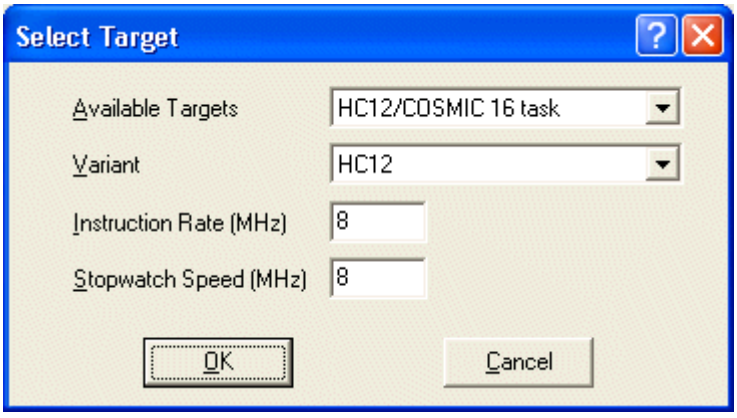


Figure 3:28 - Selecting a Target Processor

Important: The **Available Targets** list will only show the targets that have been installed on your own computer, according to your license file. Please contact LiveDevices if you cannot see the targets that you expected to. Each target may have a number of variants to reflect different chip versions based on a common processor core.

The **Select Target** dialog in Figure 3:28 can also be used to enter the **Instruction Rate** and **Stopwatch Speed**. The instruction rate should be set according to the smallest instruction cycle in the processor.

The stopwatch speed value depends on the sample rate used by timer hardware attached to the `GetStopwatch()` function that is used in Timing and Extended builds. This function is used to measure execution time in OS and application code. Ideally the stopwatch is run at the instruction rate, but on some targets this may not be possible.

When the target information has been set and the **OK** button has been clicked, the RTA-OSEK GUI automatically displays a summary of the new application. You can see this in Figure 3:6. You can refer back to this summary at any stage to see an overview of the entire system that you are creating.

Saving the Application

As soon as you create a new application, it is a good idea to save it. To save an application for the first time, from the **File** menu select **Save As...**.

In the **Save As** dialog, use the **Save In** list to navigate to the location that you want to save the file in. For the **File Name** in this example, enter the name **UserApp**. Click the **Save** button to save your new application.

You can save the application at any time by using the **File** menu to select **Save** or by pressing the **Ctrl** key and the **S** key together on the keyboard (**Ctrl+S**).

Viewing the OIL File

The file that is created is saved using OIL v2.3 syntax. This means that other OSEK compatible tools can read it. You can view the contents of the OIL file for the current application by clicking on the **View** menu and selecting **OIL File**.

You'll see the OIL file contents displayed in the lower half of the workspace, an example was shown in Figure 3:7.

In the upper part of the window you can see the details that were displayed in the workspace. In the lower part of the window you can now see the OIL file.

Additional RTA-OSEK-specific information is saved in the OIL file using comments that start with **`//RTAOILCFG`**. These comments aren't usually shown in the OIL view, but you can switch them on.

To view the RTA-OSEK-specific comments, from the **File** menu, select **Options**. The **Options** dialog opens. Select the **Show RTA Extended OIL in View** option, shown in Figure 3:29.

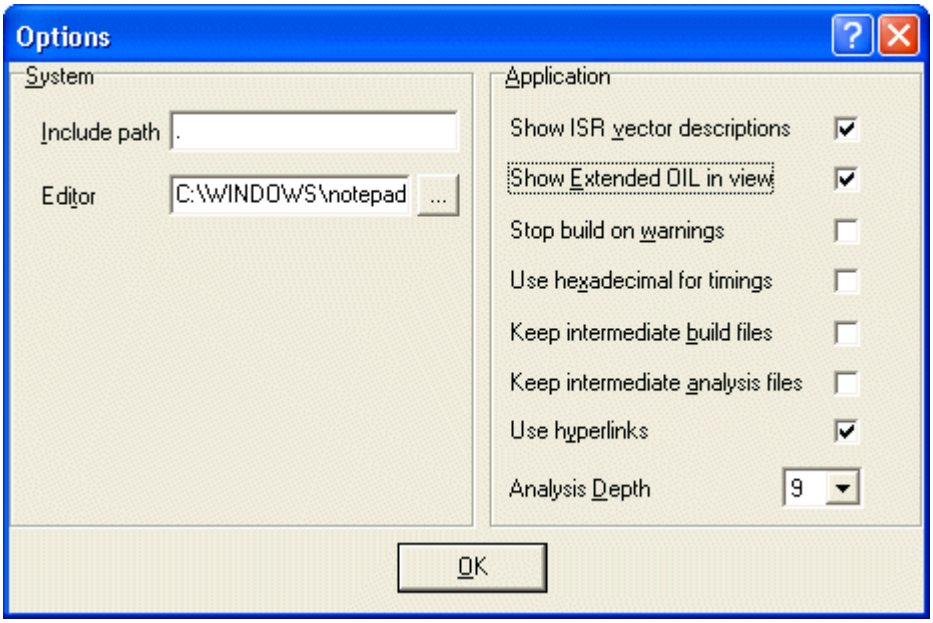


Figure 3:29 – RTA-OSEK GUI Options Window

If you click the **OK** button in the **Options** window, you'll see the comments in the OIL file.

If you are importing a legacy OIL file, comments placed within an OIL CPU object (other than those generated by RTA-OSEK) are not preserved. Comments outside the object and OIL descriptions are preserved.

You can close the OIL view by clicking on the **View** menu and deselecting **OIL File**.

Important: Do not hand-edit OIL files that use extended `//RTAOILCFG` syntax. Interdependencies exist that could cause you to lose important information when the RTA-OSEK GUI reads the file back in.

Entering Stimuli and Responses

To implement your specification, the first thing you'll need to do is to enter the stimuli and responses. Select the **Stimuli** group on the navigation bar, shown in Figure 3:30. The workspace displays the **Stimulus Summary**. Here, the summary shows that there are no stimuli in this application.

Four kinds of stimuli are available for use in an application: **bursty**, **alarm**, **periodic** and **planned**. Full details of these types of stimuli can be found in Sections 10 and 11 of this User Guide. In this example, you will create bursty stimuli to model the pressing of *Button1* and *Motor1* reaching its running state. You will also create an alarm stimulus that models *Lamp3* toggling on and off every 1s.

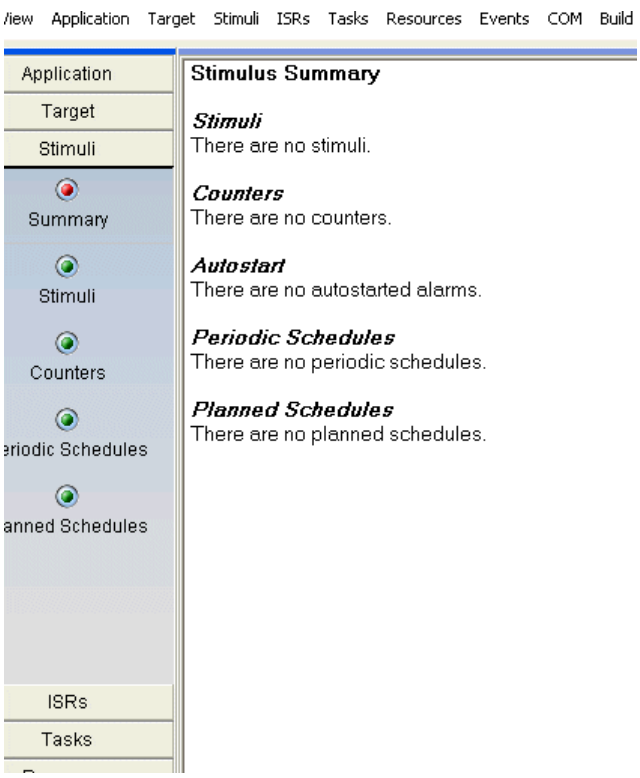


Figure 3:30 - Viewing the Stimulus Summary

To add a new bursty stimulus, select the **Stimuli** subgroup on the navigation bar. You'll see a **Select Stimulus** drop down list and three buttons.

When there are no stimuli in your application only the **Add Stimulus** button is enabled.

Click the **Add** button in the Stimulus workspace. This opens the **Add Stimulus** dialog. Enter the name ***Button1Press***.

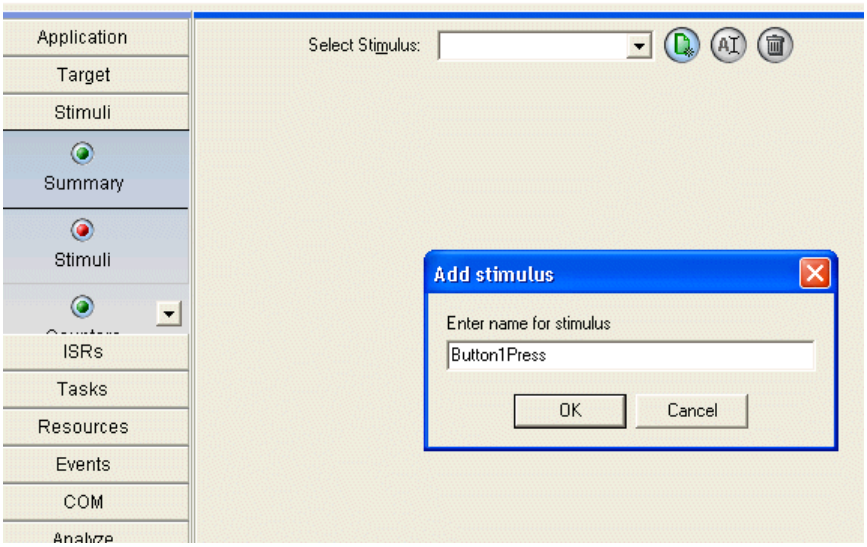


Figure 3:31 - Entering a Name for the New Stimulus

Clicking the **OK** button, as shown in Figure 3:31, creates a new stimulus. By default, the stimulus type is bursty. The default properties for this stimulus are displayed in the workspace.

Across the top of the workspace, in Figure 3:32, you will see that there is now a **Response** drop down list and all of the buttons are now enabled. The buttons that appear down the left hand side of the workspace can be used to change the stimulus properties.

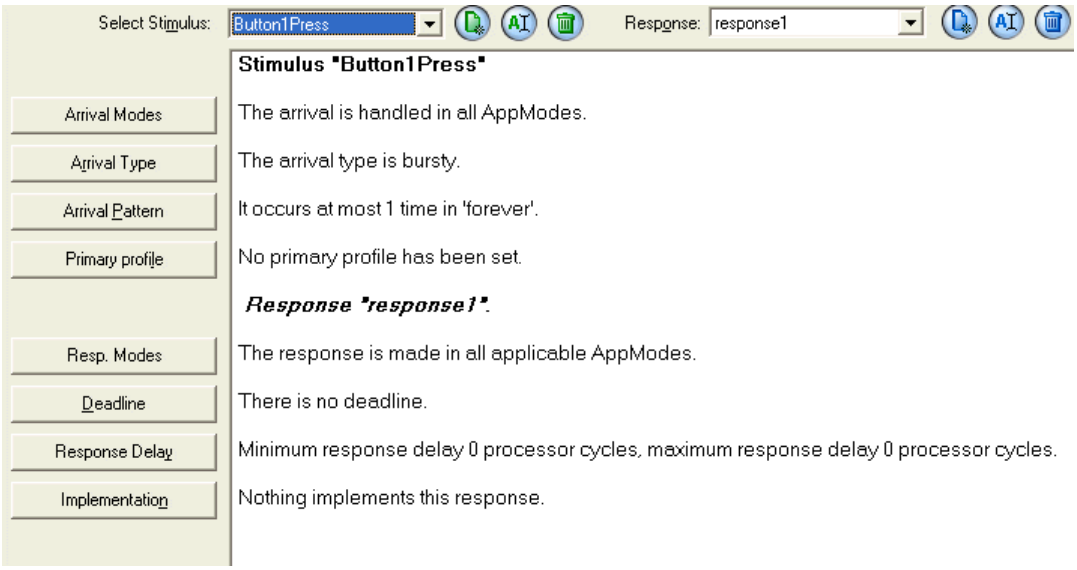


Figure 3:32 - Viewing the New Bursty Stimulus

You have now created a bursty stimulus called *Button1Press* that, by default, will only occur once. There is no primary profile set, so it will not yet be detectable. A bursty stimulus is used to model real-world events whose arrival time cannot be guaranteed, but where a maximum rate can be determined.

You will notice that the RTA-OSEK GUI also created a default response called *response1*.

The default details for this new response are shown in the workspace. At the moment the response won't have a deadline or an implementation.

Your specification says that *Button1Press* can occur many times, but no faster than twice per second, with a minimum interval of 0.1s.

To add this into your application:

- Click the **Arrival Pattern** button in the Stimulus workspace. This opens the **Arrival Burst Pattern** dialog.
- In the first row of this dialog, enter the value **1** into the **At Most...** column. Then enter **0.1** into the **In Any...** column and select **real time** and **s** from the drop down lists.

Figure 3:33 shows the information you have just entered.

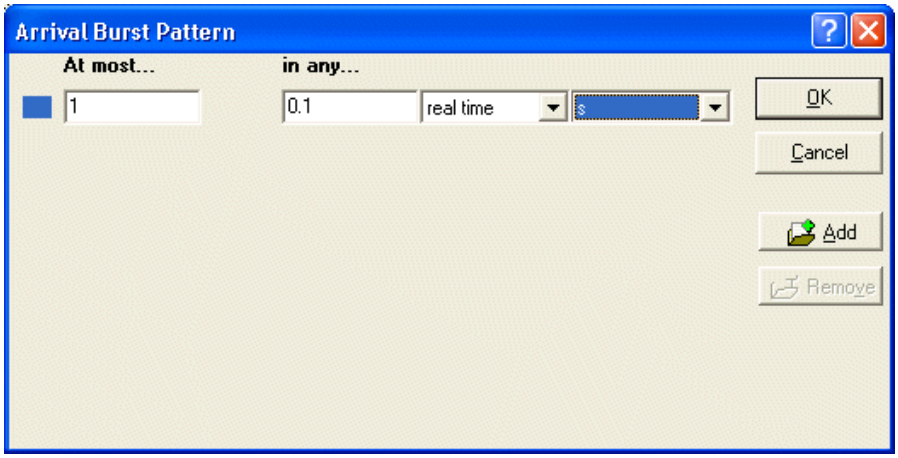


Figure 3:33 - Entering the First Arrival Burst Pattern

Now you will need to add another burst pattern of 'at most 2 times in any 1 real time s'. To add another burst pattern:

In the **Arrival Burst Pattern** dialog, click the 

button. This adds a new entry to the dialog.

- Enter **2** into the **At Most...** column. Enter **1** into the **In Any...** column and select **real time** and **s** from the drop down lists.
- Click the **OK** button to save the changes that you have made.

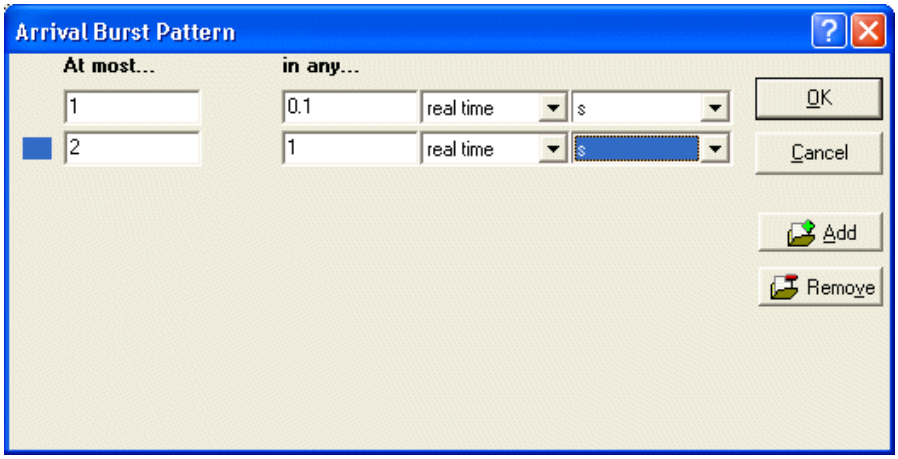


Figure 3:34 - A Stimulus with a Complex Bursting Pattern

The next part of your specification says that *Lamp1* must be lit within 10ms of *Button1Press*. This is the deadline for *Lamp1* to be lit

There is a 0.5ms delay from switching on the current to the lamp being lit.

- Click the **Rename Response** button in the Stimulus workspace. In the **Rename** dialog that opens, rename *response1* to ***Lamp1On***. Renaming the response make it clearer which response is generated when the *Button1Press* stimulus is detected.
- Click the **OK** button, as shown in Figure 3:35, to save the new name.

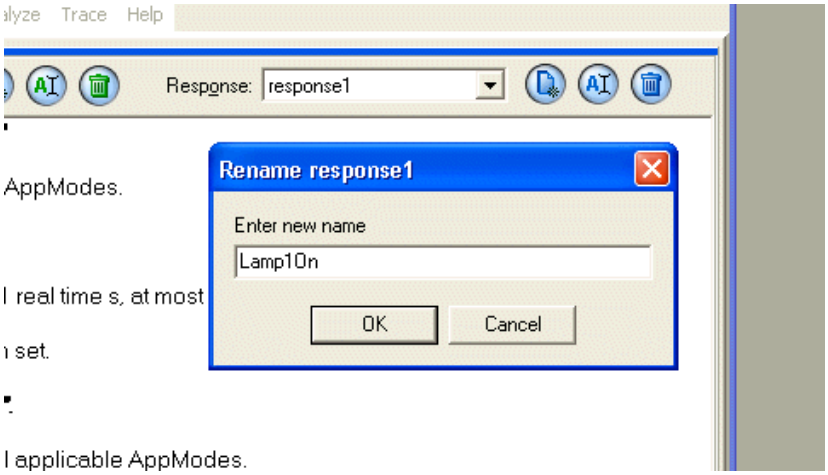


Figure 3:35 - Renaming a Response

- Click the **Deadline** button and set the deadline to **10 real time ms**, as shown in Figure 3:36.

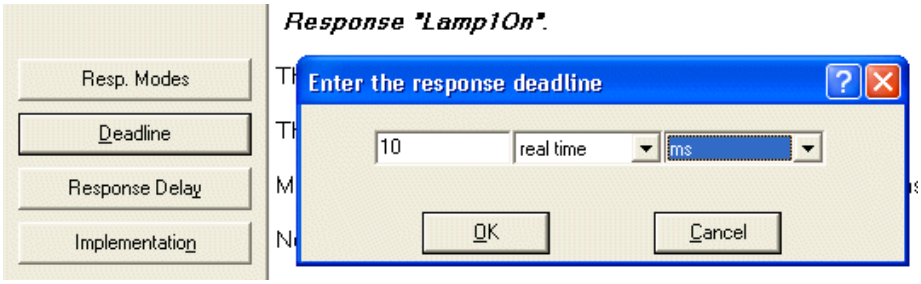


Figure 3:36 - Entering the Lamp1On Response Deadline

- Click the **Response Delay** button. Set the **Max** value to **0.5 real time ms** and the **Min** value to **0**.

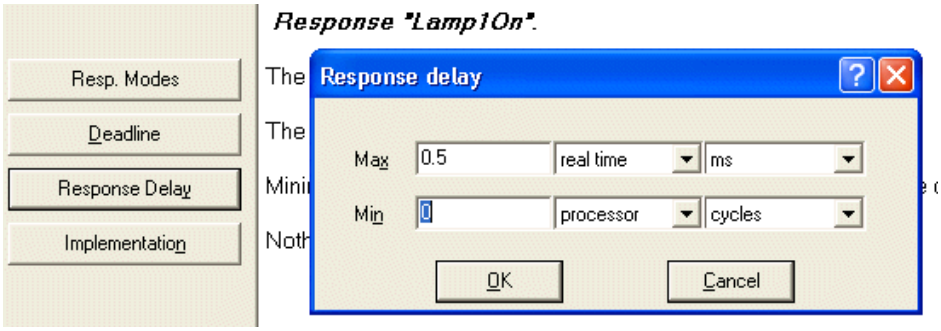


Figure 3:37 - Setting the Response Delay for the Lamp1On Response

If you look at the workspace you will see a summary of the details you have entered. Have a look at Figure 3:38.

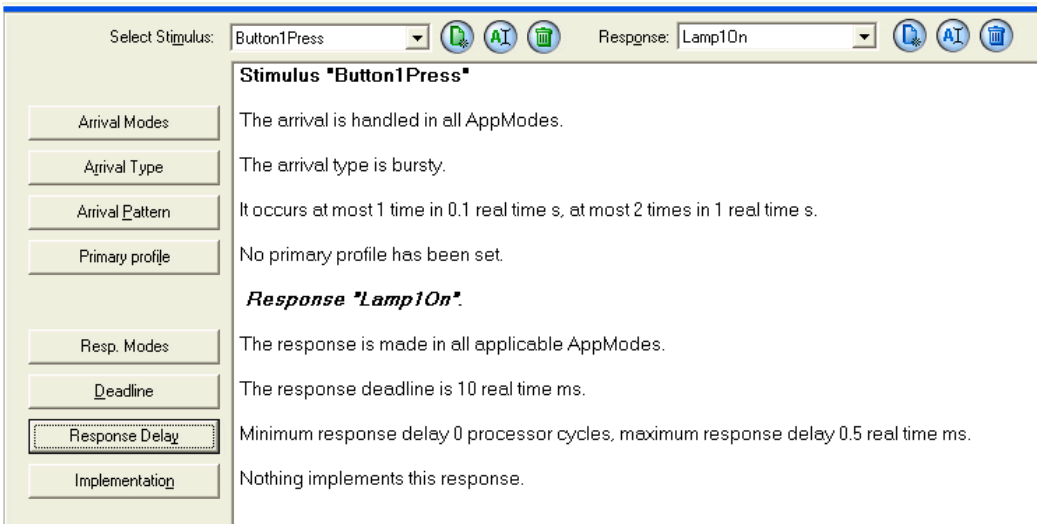

Select Stimulus:
Button1Press
Response:
Lamp1On
Stimulus "Button1Press"
Arrival Modes
The arrival is handled in all AppModes.
Arrival Type
The arrival type is bursty.
Arrival Pattern
It occurs at most 1 time in 0.1 real time s, at most 2 times in 1 real time s.
Primary profile
No primary profile has been set.
Response "Lamp1On".
Resp. Modes
The response is made in all applicable AppModes.
Deadline
The response deadline is 10 real time ms.
Response Delay
Minimum response delay 0 processor cycles, maximum response delay 0.5 real time ms.
Implementation
Nothing implements this response.

Figure 3:38 - Viewing the Button1Press Stimulus and the Lamp1On Response

The next part of the specification says that *Lamp2* must be switched off within 11ms of the *Button1Press* stimulus. It must also take into account that there is a 0.3ms delay from removing the current to the lamp being unlit.

- Add a new response by clicking the **Add** button (to the right of the **Response** drop down list).
- Enter the name ***Lamp2Off*** and click the **OK** button.
- Click the **Deadline** button and set the deadline to **11 real time ms**.
- Click the **Response Delay** button and set the **Max** value to **0.3 real time ms** and the **Min** value to **0**.

The current details are displayed in the workspace, shown in Figure 3:39.

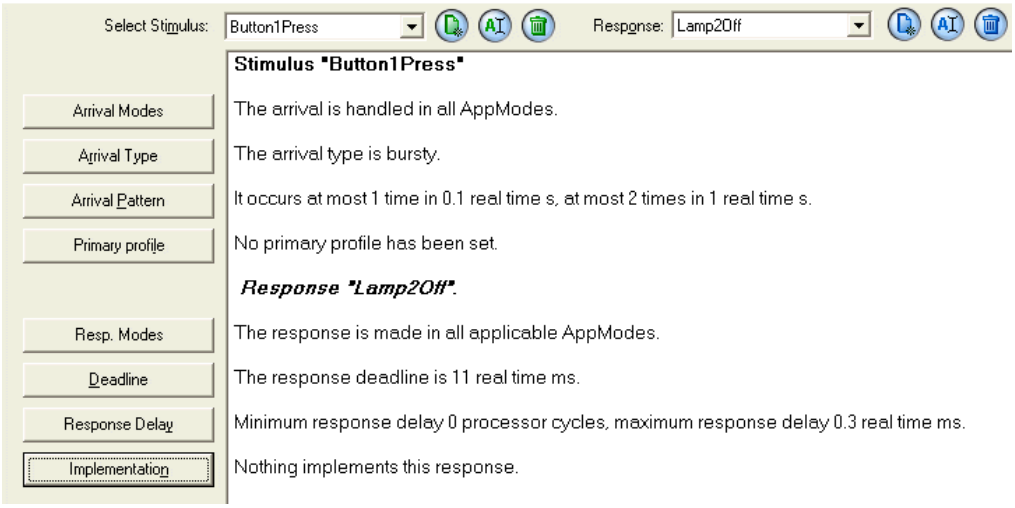

Select Stimulus:
Button1Press
Response:
Lamp2Off
Stimulus "Button1Press"
Arrival Modes
The arrival is handled in all AppModes.
Arrival Type
The arrival type is bursty.
Arrival Pattern
It occurs at most 1 time in 0.1 real time s, at most 2 times in 1 real time s.
Primary profile
No primary profile has been set.
Response "Lamp2Off".
Resp. Modes
The response is made in all applicable AppModes.
Deadline
The response deadline is 11 real time ms.
Response Delay
Minimum response delay 0 processor cycles, maximum response delay 0.3 real time ms.
Implementation
Nothing implements this response.

Figure 3:39 - Viewing the Details of the New Lamp2Off Response

Next you will need to add a response that represents *Motor1* being switched on.

- Add a new response called ***Motor1On***. Remember that you must click the **Add** button to the right of the **Response** drop down list to do this.
- Set the *Motor1On* **Deadline** to 200 real time ms.
- Set the **Response Delay Max** value to **50 real time ms** and **Min** to **0**.

The specification says that *Lamp3* has to toggle on and off every 1 second (+/-2ms). To add this to your application, you will need to add a new stimulus and a new response.

- Add a new stimulus by clicking the **Add** button (to the right of the **Select Stimulus** drop down list);
- Enter the name ***Lamp3Toggle*** and click the **OK** button.
- Click the **Arrival Type** button and set the stimulus to be periodic by selecting the ***Periodic*** option,
- Click the **Arrival Pattern** button and set the period to **1 real time s**.
- To satisfy the requirement for a toggle variation of +/-2ms, click the **Deadline** button and enter of a **4 real time ms** deadline for the response.
- Click the **Response Delay** button and set **Max** to **0.5 real time ms** (the lamp switch-on delay).

A summary of the stimuli and responses can be seen in the workspace, as shown in Figure 3:40.

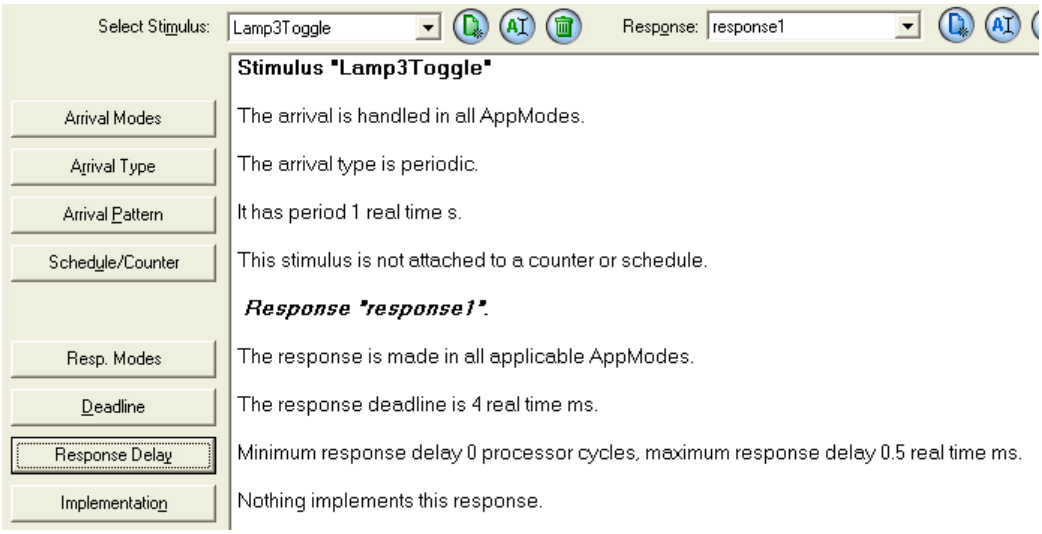


Figure 3:40 - Viewing the Details of the New Lamp3Toggle Stimulus and Response

Finally, you will need to enter the details for the motor reaching the running state.

- Add a new bursty stimulus called ***Motor1Running***.
- Rename the default response to ***Lamp2On***.
- Enter a **Deadline** of **10 real time ms**.

- Enter a maximum **Response Delay** of **0.5 real time ms**.
- Create a new response called ***Lamp1Off***.
- Specify a **Deadline** of **11 real time ms**.
- Enter a **Response Delay** of **0.3 real time ms**.

Figure 3:41 shows the details of the *Motor1Running* stimulus.

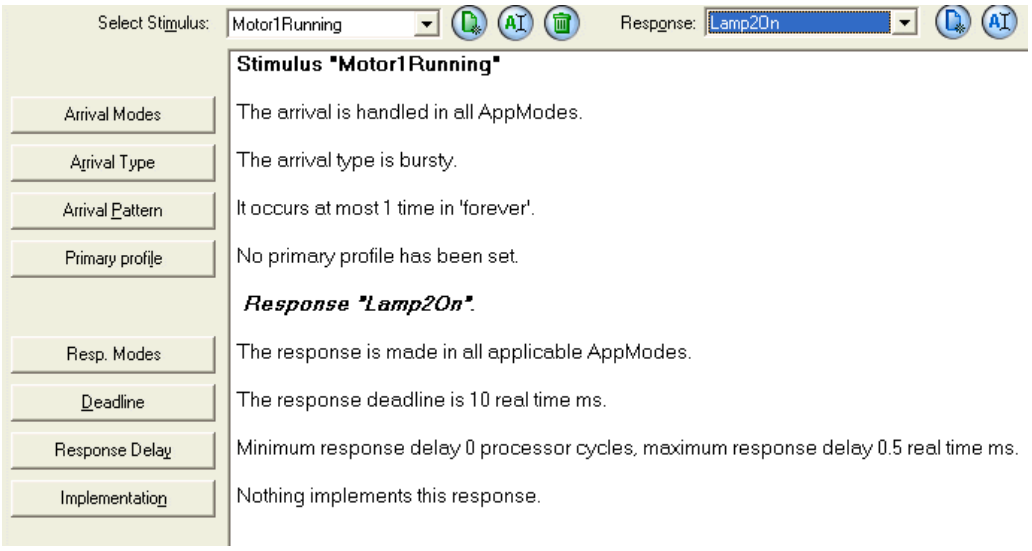


Figure 3:41 - Viewing the Details of the New Motor1Running Stimulus

The information provided in the specification has now been entered. Have a look at the Stimulus Summary to see an outline of the details you have specified. The workspace should appear as shown in Figure 3:42.

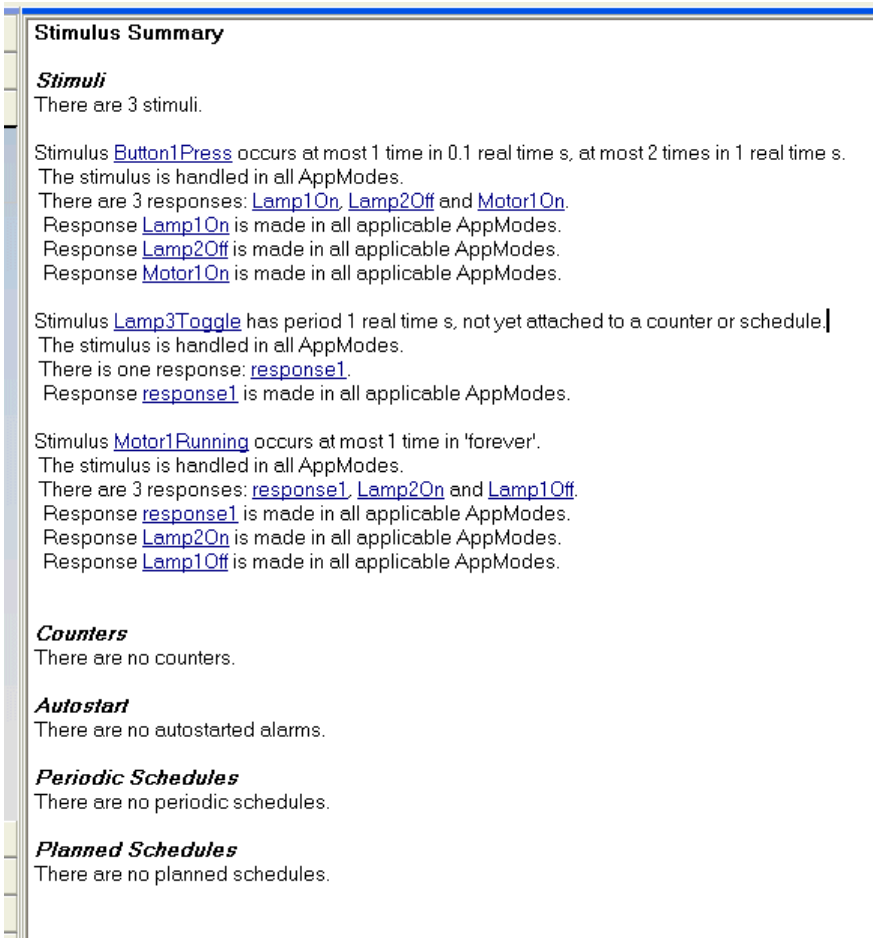

Stimulus Summary
Stimuli
There are 3 stimuli.
Stimulus Button1Press occurs at most 1 time in 0.1 real time s, at most 2 times in 1 real time s.
The stimulus is handled in all AppModes.
There are 3 responses: Lamp1On, Lamp2Off and Motor1On.
Response Lamp1On is made in all applicable AppModes.
Response Lamp2Off is made in all applicable AppModes.
Response Motor1On is made in all applicable AppModes.
Stimulus Lamp3Toggle has period 1 real time s, not yet attached to a counter or schedule.
The stimulus is handled in all AppModes.
There is one response: response1.
Response response1 is made in all applicable AppModes.
Stimulus Motor1Running occurs at most 1 time in 'forever'.
The stimulus is handled in all AppModes.
There are 3 responses: response1, Lamp2On and Lamp1Off.
Response response1 is made in all applicable AppModes.
Response Lamp2On is made in all applicable AppModes.
Response Lamp1Off is made in all applicable AppModes.
Counters
There are no counters.
Autostart
There are no autostarted alarms.
Periodic Schedules
There are no periodic schedules.
Planned Schedules
There are no planned schedules.

Figure 3:42 - Reviewing the Stimulus Summary

Remember that you should always save your application regularly, by selecting **Save** from the **File** menu.

3.3.2 Implementation

You have now reached the implementation stage. It is now time to decide how the stimuli are detected and how responses are implemented. You will also learn how to create ISRs and tasks using the RTA-OSEK GUI.

Creating an ISR

You will need three interrupt sources in this example. One that detects the button press, another attached to a hardware timer that can provide interrupts every 100ms and the third attached to the motor.

In this example, let's assume that all three interrupt sources can be serviced by a single ISR. This ISR will be called *PrimaryISR*.

To create a new ISR, select the **ISRs** group from the navigation bar.

Figure 3:43 shows the how the ISRs group is selected and how the ISR Summary initially appears in the workspace.

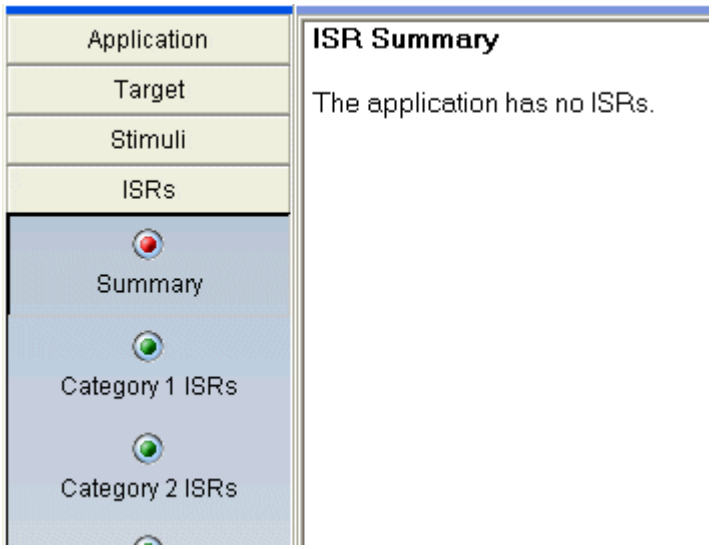


Figure 3:43 - Selecting the ISRs Group from the Navigation Bar

In this example you must use a Category 2 ISR because you will need to activate tasks. Category 1 ISRs are not allowed to make OS API calls, so they cannot be used in this case.

- From the navigation bar, select the **Category 2 ISRs** subgroup.
- Create the ISR by clicking the **Add** button in the workspace.
- In the **Add Cat 2 ISR** dialog, enter the name ***PrimaryISR*** and click **OK**.
- Depending on the target type, you may need to enter an interrupt **Vector** and a **Priority**. An example is shown in Figure 3:44.

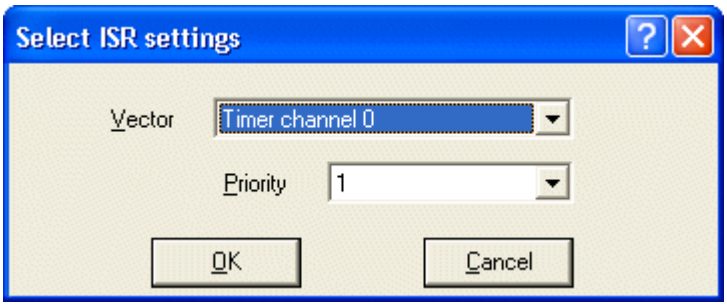


Figure 3:44 - Selecting the ISR Vector and Priority

The workspace displays the default settings for the new ISR.

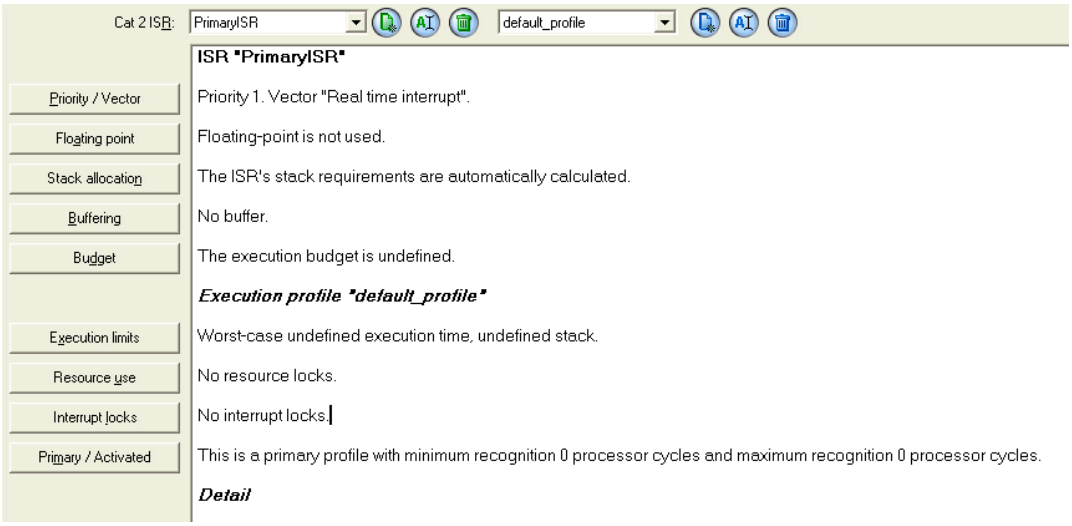


Figure 3:45 - Viewing the Details of the New PrimaryISR

Notice that, at the top of the workspace, the RTA-OSEK GUI automatically created an execution profile called *default_profile* for the new ISR.

Execution profiles are used to describe different paths of execution through a task or ISR for timing analysis purposes. The ISR, in this example, has to establish which interrupt sources are pending, so that it can react to the correct stimulus.

For the moment, the ISR will exit after reacting to an interrupt source rather than checking the other sources. The execution path taken by the ISR can, therefore, take one of three paths. The code will look something like Code Example 3:5.

```
#include "PrimaryISR.h"
ISR(PrimaryISR)
{
  if (Button1PressInterruptPending()) {
    /* Button1Press detected. */
  } else if (Motor1RunningPending()) {
    /* Motor1Running detected. */
  } else {
    /* Timer expiry detected. */
  }
}
```

Code Example 3:5 - Paths of Execution for an ISR

Three execution profiles must be created.

- Rename the existing execution profile by clicking the **Rename** button (to the right of the execution profile drop down list). This is can be seen in Figure 3:46.

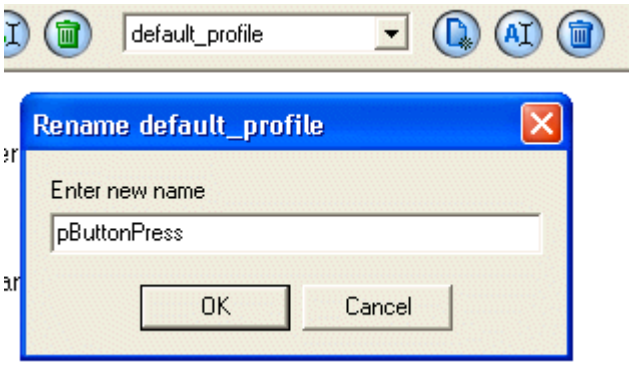


Figure 3:46 - Renaming an Execution Profile

- In the **Rename** dialog, change the name of the execution profile from *default_profile* to *pButtonPress*.

You can now enter the time allowance for the *Button1* debounce circuitry. Recognition time only applies to primary profiles. It is the min/max time between a real-world event occurring and the resulting state change happening at the processor. Recognition time is an important value for timing analysis, particularly in distributed systems.

- Click the **Primary/Activated** button. The **Primary or Activated Profile** dialog opens.
- Set the interrupt **Recognition Time** to **0.4 real time ms** for the **Max** value and **0.1 real time ms** for **Min**. This is shown in Figure 3:47.
- Click the **OK** button.

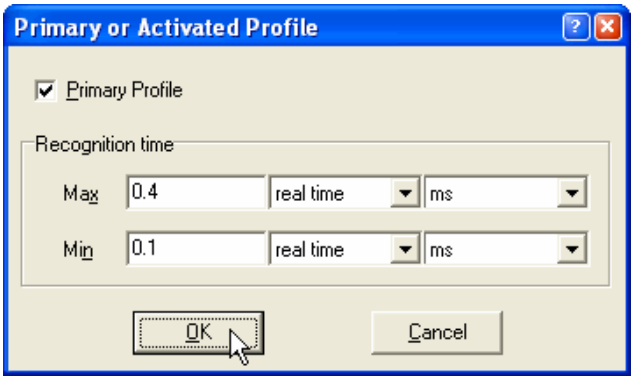


Figure 3:47 - Entering the Primary Profile Settings

- Create a new profile by clicking the **Add** button (to the right of the execution profiles drop down list) as shown in Figure 3:48.



Figure 3:48 - Adding a New Execution Profile

- In the **Add execution profile** dialog, enter the name *pMotor1Running*.
- Now create another new profile called *pTimer*.

You must now tell the RTA-OSEK GUI that the three new profiles reflect individual sources and that only one is processed at a time.

This is achieved by informing the RTA-OSEK GUI that the ISR is retriggering. This means that one interrupt is handled at a time by the ISR. The ISR then returns and pending interrupts will retrigger the ISR.

- In the ISR workspace, click the **Buffering** button to launch the **Specify ISR buffering behavior** dialog box as shown in Figure 3:49. Clear the **Simple – no buffering** check box and set the ISR's buffering to **Retrigger after Leaving ISR** and **Buffer by Execution Profile**.

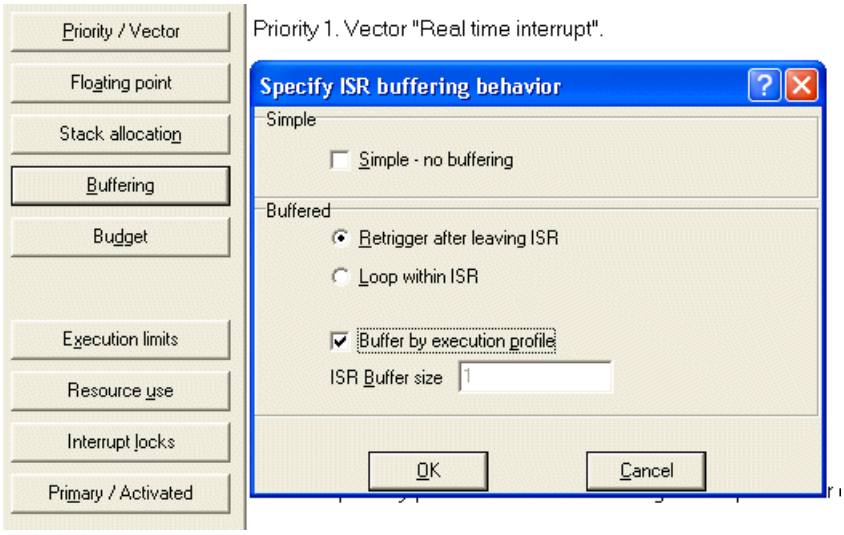


Figure 3:49 - Specifying ISR Buffering Behavior for PrimaryISR

You can see a summary of the ISR by selecting the **Summary** subgroup, as shown in Figure 3:50.

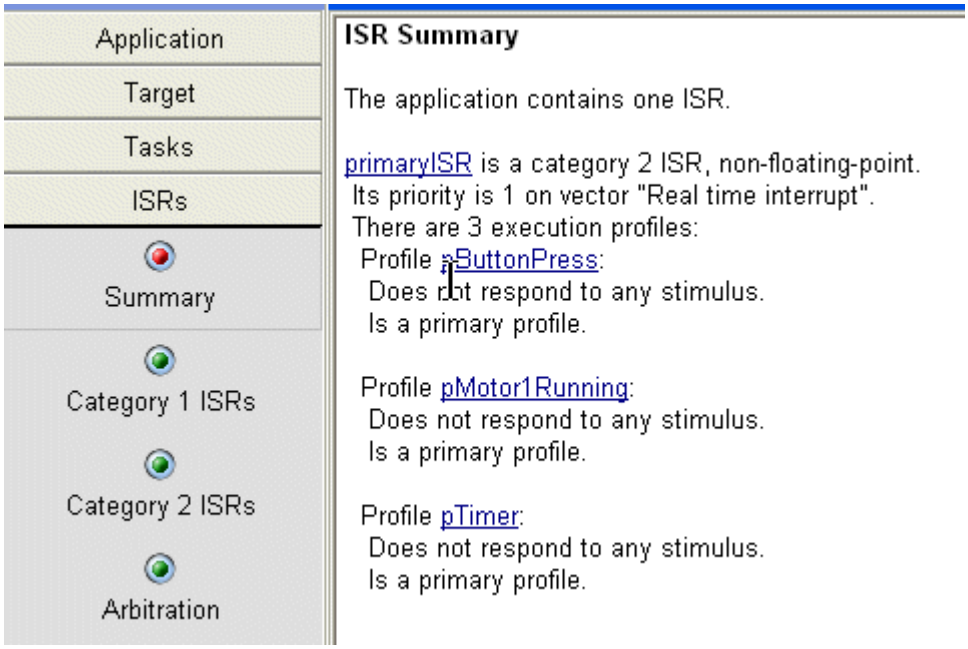


Figure 3:50 - Viewing the ISRs Summary

Attaching an ISR to a Stimulus

The new ISR needs to be attached to the *Button1Press* and *Motor1Running* stimuli that you created earlier. The RTA-OSEK GUI will then know that it will be responsible for generating the responses required when the stimuli are detected.

- From the navigation bar, select the **Stimuli** group.
- Select the **Stimuli** subgroup and then use the **Select Stimulus** drop down list to select ***Button1Press***.
- Click the **Primary Profile** button. The Select Profile dialog opens.
- From the Primary Profiles drop down list, select ***PrimaryISR.pButtonPress*** and click the OK button, as shown in Figure 3:51.

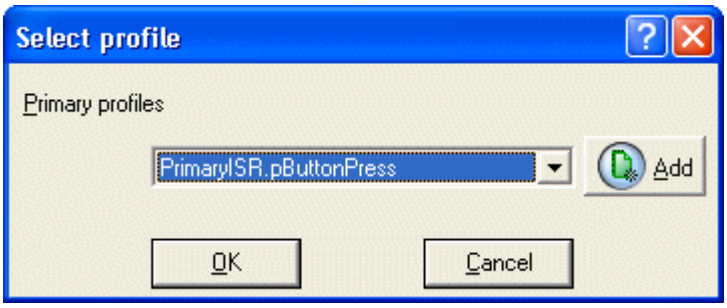


Figure 3:51 - Selecting a Primary Profile for Button1Press

You have now shown that the profile *pButtonPress* of ISR *PrimaryISR* is responsible for reacting to stimulus *Button1Press*.

- Use the **Select Stimulus** drop down list to select the ***Motor1Running*** stimulus.

- Click the **Primary Profile** button. In the **Select Profile** dialog, select ***PrimaryISR.pMotor1Running*** and then click the **OK** button.

You will still need to do some more configuration before you can use the *pTimer* profile; remember that the timer interrupts every 100ms, but the *Lamp3* toggle only occurs every 1s. You will need something that will count each interrupt tick and raise the stimulus *Lamp3Toggle* every 10 ticks. This can be achieved simply by using an OSEK counter object.

- From the **Stimuli** group on the navigation bar, select the **Counters** subgroup.
- Click the **Add** button to create a new counter called ***TimerCounter***.
- When prompted, specify that the **Fastest Tick Rate** is **100 real time ms**.

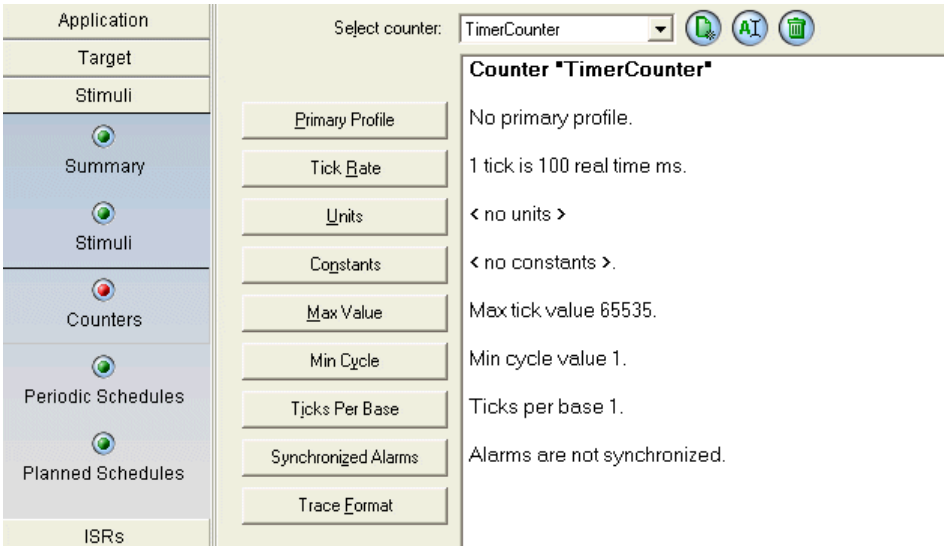


Figure 3:52 - Viewing the Default Details for the new Counter

- Click the **Primary Profile** button and use the **Primary Profiles** drop down list to select ***PrimaryISR.pTimer***.
- Now, from the navigation bar, go to the **Stimuli** group and select stimulus ***Lamp3Toggle***.
- On the **Counter** tab, click the **Schedule/Counter** button and specify that the stimulus is attached to counter ***TimerCounter***. This is shown in Figure 3:53. (If you are familiar with OSEK concepts, you will recognize that this stimulus has now been implemented as an OSEK alarm.)

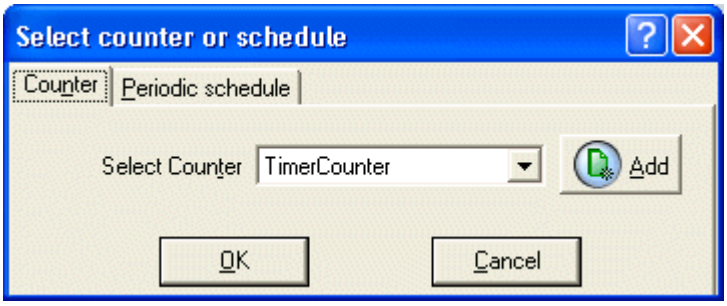


Figure 3:53 - Selecting a Counter for Lamp3Toggle

In Figure 3:54, you can see that the Stimulus Summary shows that all three stimuli are now driven by the appropriate primary profiles.

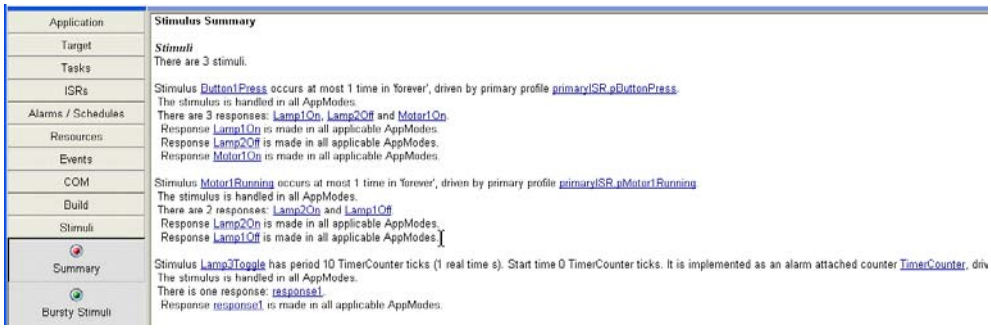

Application
Target
Tasks
ISRs
Alarms / Schedules
Resources
Events
COM
Build
Stimuli
Summary
Bursty Stimuli
Stimulus Summary
Stimuli
There are 3 stimuli.
Stimulus Button1Press occurs at most 1 time in 'forever', driven by primary profile primaryISR.pButtonPress.
The stimulus is handled in all AppModes.
There are 3 responses: Lamp1On, Lamp2Off and Motor1On.
Response Lamp1On is made in all applicable AppModes.
Response Lamp2Off is made in all applicable AppModes.
Response Motor1On is made in all applicable AppModes.
Stimulus Motor1Running occurs at most 1 time in 'forever', driven by primary profile primaryISR.pMotor1Running
The stimulus is handled in all AppModes.
There are 2 responses: Lamp2On and Lamp1Off.
Response Lamp2On is made in all applicable AppModes.
Response Lamp1Off is made in all applicable AppModes.
Stimulus Lamp3Toggle has period 10 TimerCounter ticks (1 real time s). Start time 0 TimerCounter ticks. It is implemented as an alarm attached counter TimerCounter, driv
The stimulus is handled in all AppModes.
There is one response: response1.
Response response1 is made in all applicable AppModes.

Figure 3:54 - Viewing the Primary Profiles on the Stimulus Summary

Creating Responses

You saw earlier that responses are normally implemented using tasks. In this example you will need 4 tasks:

1. Task *Button1Response* will be used to implement both responses *Lamp1On* and *Lamp2Off* when *Button1* is pressed.
2. Task *MotorStart* will be used to implement response *MotorOn* when *Button1* is pressed.
3. Task *LampToggle* will be used to implement response *Lamp3Toggle* when the alarm *Lamp3Toggle* occurs.
4. Task *MotorResponse* will be used to implement both responses *Lamp2On* and *Lamp1Off* when the motor runs.

Looking at the deadlines involved, *LampToggle* should have the highest priority, because it is associated with the shortest deadline. *MotorStart* can be given the lowest priority, because its associated deadline is the longest. The other two tasks can be given 'medium' priorities.

Let's start with the responses for stimulus *Button1Press*.

- Select stimulus ***Button1Press*** and response ***Lamp1On***.

Click the **Implementation** button and then click the **Add** button.

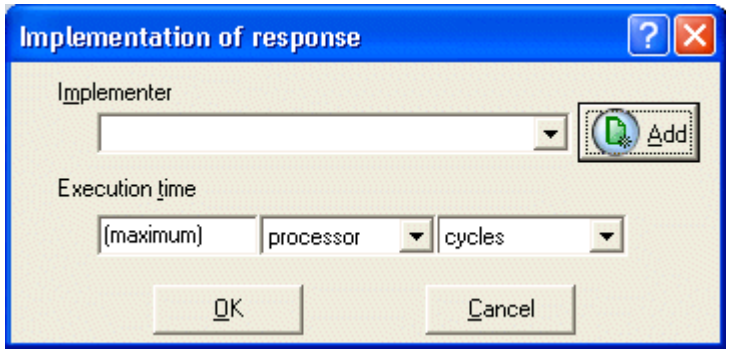


Figure 3:55 - Creating a Task or ISR from the Implementation of Response Dialog

The **Create Task or ISR** dialog opens (Figure 3:56).

- Select the **Task** option and click the **OK** button.

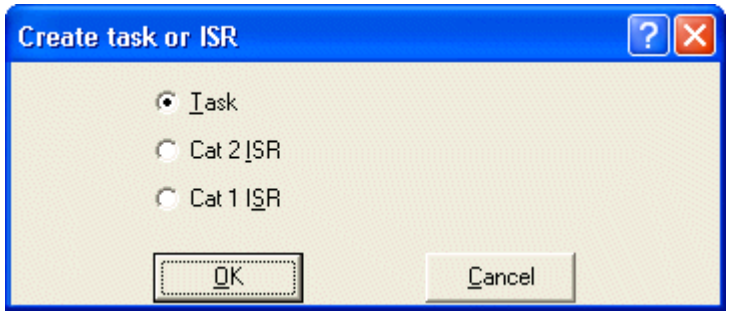


Figure 3:56 - Creating a New Task

This opens the **Add Task** dialog.

- Create a task named ***Button1Response***. Assign it priority **10** and click OK.
- The execution profile can be left as ***default_profile***. At this stage, there is no need to specify an execution time. Click OK, then Ok again.



Figure 3:57 - Viewing the Implementation Details for the Lamp1On Response

- Now make sure that the **Select Stimulus** drop down list has ***Button1Press*** selected and use the **Response** drop down list to select ***Lamp2Off***.
- Click the **Implementation** button. You can simply select ***Button1Response*** from the execution profile list because this time you are going to implement the response in the task that you just created. Click OK.

- From the **Response** drop down list in the workspace, select the ***Motor1On*** response.
- Click the **Implementation** button to create a new task ***MotorStart*** with priority ***5***. Once again, there is no need to rename the ***default_profile*** and the execution time can be left undefined.

Next add the response for stimulus *Lamp3Toggle*.

- Select the stimulus ***Lamp3Toggle***.
- Click the **Implementation** button and add a new task called ***LampToggle*** with priority **20**.

Finally you can add the responses for stimulus *Motor1Running*.

- Select stimulus ***Motor1Running*** and response ***Lamp2On***.
- Click the **Implementation** button and add a new task ***MotorResponse*** with priority ***9***.
- Select response ***Lamp1Off***.
- Click the **Implementation** button and select task ***MotorResponse***.

The **Stimulus Summary** now shows all three stimuli and the primary profile. The configuration of the application is complete.

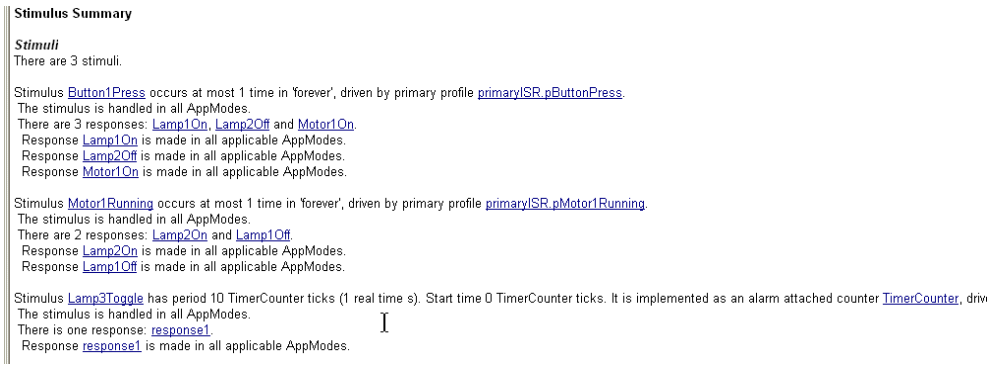

Stimulus Summary
Stimuli
There are 3 stimuli.
Stimulus Button1Press occurs at most 1 time in 'forever', driven by primary profile primaryISR.pButtonPress.
The stimulus is handled in all AppModes.
There are 3 responses: Lamp1On, Lamp2Off and Motor1On.
Response Lamp1On is made in all applicable AppModes.
Response Lamp2Off is made in all applicable AppModes.
Response Motor1On is made in all applicable AppModes.
Stimulus Motor1Running occurs at most 1 time in 'forever', driven by primary profile primaryISR.pMotor1Running.
The stimulus is handled in all AppModes.
There are 2 responses: Lamp2On and Lamp1Off.
Response Lamp2On is made in all applicable AppModes.
Response Lamp1Off is made in all applicable AppModes.
Stimulus Lamp3Toggle has period 10 TimerCounter ticks (1 real time s). Start time 0 TimerCounter ticks. It is implemented as an alarm attached counter TimerCounter, driv
The stimulus is handled in all AppModes.
There is one response: response1.
Response response1 is made in all applicable AppModes.

Figure 3:58 - Viewing the Stimulus Summary with Primary Profile and Responses

Writing Task and ISR Code

You will now need code for the ISR *PrimaryISR*, tasks *Button1Response, MotorStart*, *LampToggle* and *MotorResponse*, as well as the application's `main()` function (includes application startup and idle mechanism).

The C source code can be created externally from the RTA-OSEK GUI if you wish, but you can also create templates from the RTA-OSEK GUI to help get you started. To do this:

- Change to the *Builder*, then from the navigation bar, select the **Custom Build** group.
- In the workspace, click the **Create Templates** button. RTA-OSEK will create seven C source files and put skeleton code in each of them. It

also creates a batch file `rtkbuild.bat`, which you will use later in the build phase.

Writing Code for PrimaryISR

From the navigation bar, select the **ISRs** group. Then select the **Category 2 ISRs** subgroup and from the workspace, select ***PrimaryISR***.

You don't have to worry if you can't remember everything that has to be done in the ISR, because the RTA-OSEK GUI can tell you. Simply select the **Implementation** option from the **View** menu. The lower section of the ISR window is displayed and the implementation details for the ISR will appear. You can resize this window by moving your mouse over the blue horizontal 'splitter' bar. Click and hold the left mouse button and drag the bar up or down.

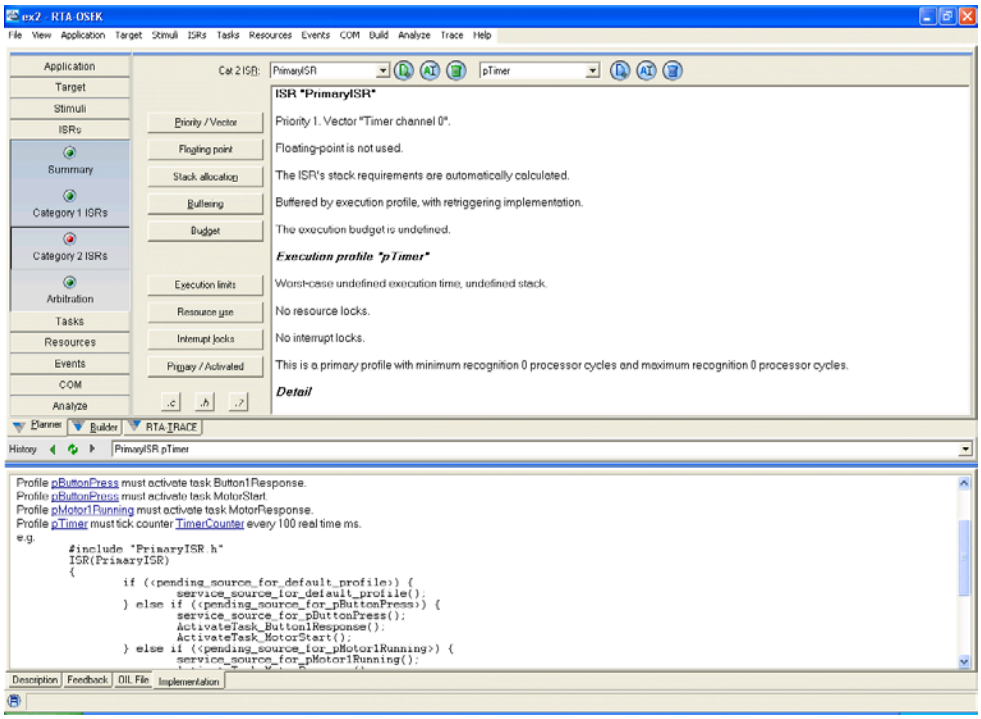


Figure 3:59 - Viewing the PrimaryISR Implementation Notes

The sample code shows the ISR-specific header file `PrimaryISR.h` being `#included`, followed by the ISR body. The three execution profiles are reflected in the three routes through the `if...else...else` construct.

Notice that the tasks that provide the responses are activated in the appropriate execution profiles and counter *TimerCounter* is ticked in the *pTimer* profile.

You can close the implementation notes by deselecting the **Implementation** option in the **View** menu.

Important: You must implement the flow of control exactly as shown by the RTA-OSEK GUI. If you don't do this, the system you implement will have

different timing characteristics from the system that RTA-OSEK will later use to perform timing analysis. **In particular, do not re-arrange the order of checking for interrupt sources and do not loop back to test for any interrupts that are still pending without specifying that the ISR has 'looping' behavior.**

You can directly edit the source code for the ISR by selecting the  button in the Category 2 ISRs workspace.

Important: When editing files from within the RTA-OSEK GUI, the default editor is set to be the Windows Notepad application. You can select your own preferred editor from the **File** menu, by selecting **Options**.

Portability note: The code that needs to be written here is target-specific, so no details are given where lines involve detection of pending interrupt sources and how they are acknowledged.

Writing Code for Button1Response

From the **Tasks** group on the navigation bar, select the **Task Data** subgroup. Then select the task ***Button1Response***.

Use the **Implementation View** to check the code that is required for this task.

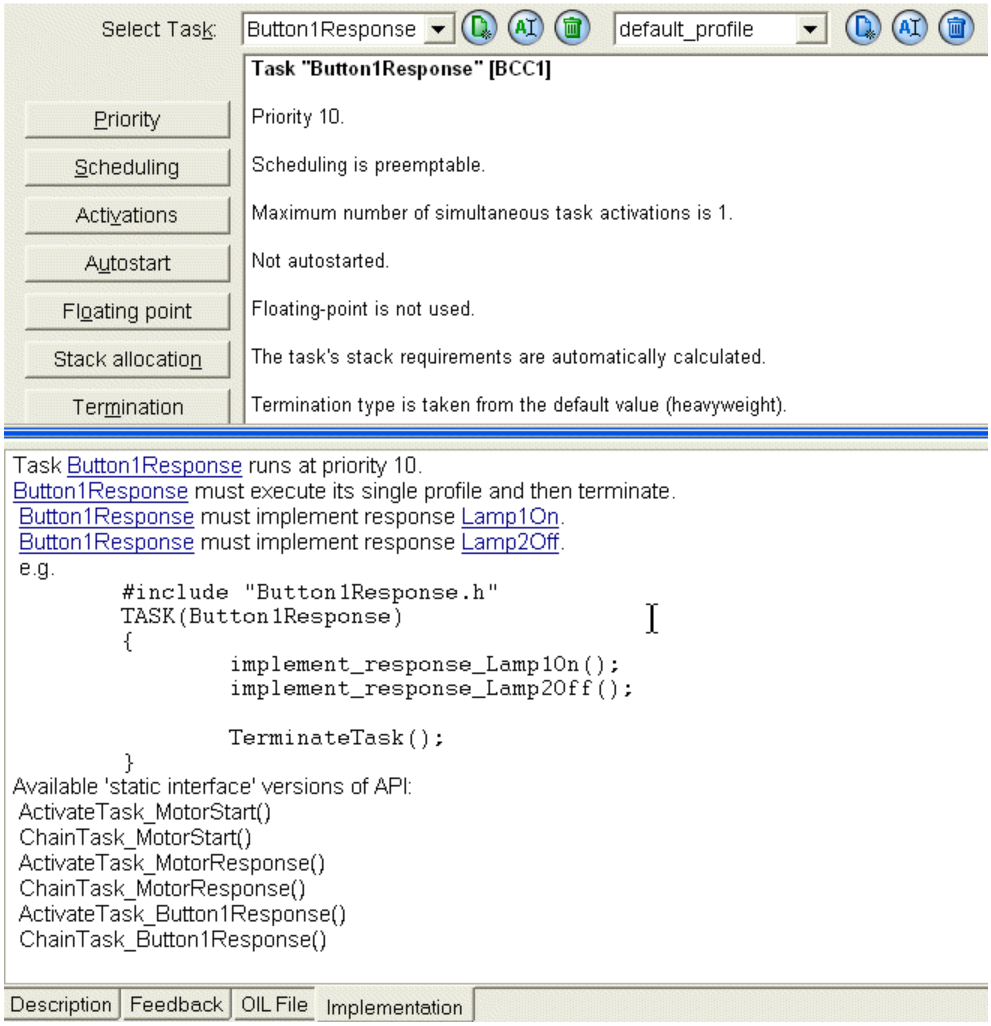


Figure 3:60 - Viewing the Implementation Notes for Button1Response

You can directly edit the source code for the task by selecting the .c button in the workspace. The code you write will be target-specific, but should follow the structure in the implementation view.

Writing Code for the Remaining Tasks

The code for tasks *LampToggle*, *MotorResponse* and *Motor1On* follow the same pattern.

Whenever you modify the RTA-OSEK configuration always check that the suggested implementation matches the code you have written.

Writing Code for 'main'

In C programs, `main()` is the starting point for the main application. It is called after the low-level startup initialization. Usually interrupts are disabled prior to `main()` being entered.

The skeleton code generated by the RTA-OSEK for `main()` is shown in Code Example 3:6.

```
/* Template code for 'main' in project: UserApp */

#include "osekmain.h"

OS_MAIN()
{
  StartOS(OSDEFAULTAPPMODE);
  ShutdownOS(E_OK);
}
```

Code Example 3:6 - Template Code for main()

Note that the `OS_MAIN()` macro is used rather than `main()`. Individual compilers have different criteria for the arguments and return types that are allowed for `main()`, so RTA-OSEK provides `OS_MAIN()` to assist portability.

Portability: Using `OS_MAIN()`, rather than `main()`, in applications can help make them more portable to different RTA-OSEK targets.

The `StartOS(OSDEFAULTAPPMODE)` call is used to start the operating system. No operating system API calls should be made before `StartOS()` is called.

The `ShutdownOS()` call is used to stop the OS when (and if) the application completes. The default action for `ShutdownOS()` is to stay in an infinite loop and not to return. This call is not normally used because applications tend to run 'forever' (or until the processor loses power or is reset).

In your example application, you need to perform some initialization of the target hardware before calling `StartOS()`. This makes sure that the timer is set to interrupt every 100ms and the appropriate interrupt sources are enabled. Then, after `StartOS()`, the alarm should be enabled and an idle loop should be entered.

In fact, the code that executes after `StartOS()` belongs to the idle task. The idle task is called `osek_idle_task`.

The idle task can act like any other task; it can make API calls, use resources, send and receive messages, send and wait for events and so on. It cannot be directly activated because it only terminates when `ShutdownOS()` is called and it cannot use internal resources because it would prevent other tasks from starting.

Important: Putting code in the idle task can be a very efficient way of implementing a system. In particular, if you have only one task that needs to respond to OSEK events you should use the `osek_idle_task`. Your application will be significantly smaller and more responsive if the idle task waits for events, rather than any other task.

The RTA-OSEK GUI will show you a suggested implementation for `OS_MAIN()`. Select the ***osek_idle_task*** task in the **Task Data** subgroup (in the **Tasks** group on the navigation bar) and view the implementation details.

Note that in this case, the idle task does no work. On targets that support it, you can put the processor into a 'sleep' state in the idle task. The processor must 'wake-up' if an interrupt occurs.

Important: The idle task must not terminate. It must loop forever.

Setting up Timer/Counter Hardware

In Code Example 3:7, the function `do_target_initialization()` needs to initialize the interrupt sources. One of these sources is a hardware counter/timer that needs to provide an interrupt every 100ms.

You may wish to use code based on Code Example 3:7 to do this.

```
void do_target_initialization(void)
{

  unsigned int timer_divide;
  timer_divide = OSTICKDURATION_TimerCounter /
OS_NS_PER_CYCLE;

  /* Target specific. */
  SetupTimer(timer_divide);
  EnableTimerInterrupt();
  EnableKeyPressInterrupt();

  /* Set up Button1 and Motor1 interrupts. */
  ...

}
```

Code Example 3:7 - Initializing Timer Hardware

Code Example 3:7 shows initialization using the two RTA-OSEK-generated constants `OSTICKDURATION_TimerCounter` and `OS_NS_PER_CYCLE`.

The `OSTICKDURATION_TimerCounter` constant specifies the duration of the 'tick' of the counter in nanoseconds (ns), so in this example the `OSTICKDURATION` is 100,000,000ns (1/10th of a second).

The `OS_NS_PER_CYCLE` constant specifies the duration of the CPU instruction cycle in ns. For a 10MHz CPU, this is 100ns.

In this example, you require the timer to be configured to interrupt every 1,000,000 instruction cycles. If you use these constants to calculate the divide ratio, the code will automatically adjust if the clock rate changes.

OS Status, ErrorHook and Callbacks

For preliminary testing, you should run the application using the operating system's **Extended** build. Extended build means that the OS performs rigorous checks in each API call. Of course, this takes time and code space.

Once the application is seen to be working correctly, you will usually switch to **Standard** build. Very few checks are made with Standard build, so the OS can run much more efficiently.

When using Extended build, you can check the return status code from each API call or alternatively request that Error Hook be used. This is a function that the OS will call whenever an error is detected. You write the implementation of Error Hook in your application. Normally you will use it to halt debugging and to alert you of errors.

To use the Error Hook facility

- Select the Application group from the navigation bar and then select the OS Configuration subgroup.
- In the **OS Configuration Summary** workspace, click the **Hooks** button. The **Select Hooks** dialog opens.
- Select the **Error Hook** checkbox and then click the **OK** button.

You can see that the Error Hook has been selected in Figure 3:61.

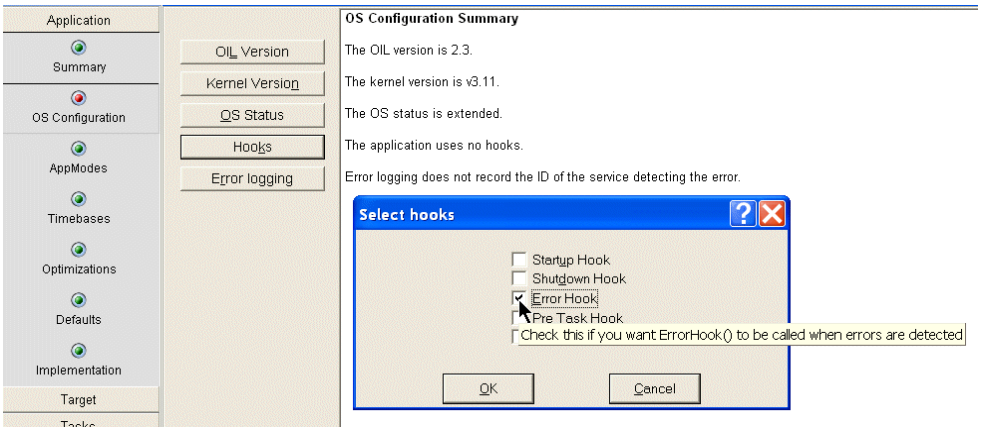


Figure 3:61 - Selecting the Error Hook

You can add the code needed to implement `ErrorHook()` in any source file, but `main.c` is a good place to start. Add the following code:

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType e)
{
  /* Put a debugger breakpoint here. */
  while (1) {
        /* Freeze. */
  }
}
#endif /* OSEK_ERRORHOOK */
```

Code Example 3:8 - The ErrorHook()

You can find more information about using `ErrorHook()` for debugging purposes in Section 13 of this User Guide.

There are three other functions that you must supply when using the Timing or Extended build. The operating system uses these to time the execution of your code.

Don't worry about the details at the moment; simply add the code in Code Example 3:9 after the `ErrorHook()`.

```
#ifdef OS_ET_MEASURE
OS_HOOK(void) OverrunHook(void)
{
  /* Put a debugger breakpoint here. */
  while (1) {
    /* Freeze. */
  }
}
OS_NONREENTRANT(StopwatchTickType)
GetStopwatch(void)
{
  /* Temporary implementation. A correct solution
   * returns the current stopwatch value. */
  return 0;
}
OS_NONREENTRANT(StopwatchTickType)
GetStopwatchUncertainty(void)
{
  /* Temporary implementation. A correct solution
   * returns the uncertainty in the stopwatch
   * value. */
  return 0;
}
#endif /* OS_ET_MEASURE */
```

Code Example 3:9 - Timing Callbacks

Final Checks

To view a complete implementation summary, from the **Application** group on the navigation bar, select the **Implementation** subgroup.

Use this as a checklist to ensure that your application is fully implemented. You can print out this summary by selecting **Print Selection** from the **File** menu.

3.3.3 Build

If you have successfully completed all of the steps in creating this example application, you can now start the build process. Switch to the *Builder*, and refer to Section 3.5.2 where the build process is described.

3.3.4 Functional Testing

The executable file can be downloaded to your target hardware, so that you can test its behavior.

Initial testing should always be performed using the Extended build with the Error Hook, because the OS will detect any misuse of API calls. Only once an application performs correctly should you switch to the Timing or Standard builds.

3.3.5 Analysis

At this stage your application appears to work, but how do you know that it meets all of its deadlines every time? If you do not use timing analysis you cannot be sure that there is no a rare combination of circumstances that will cause deadlines to be missed. Even with thousands of hours of testing, you may not pick up that 'once in a million' failure.

Later in this guide, you will find out how to measure the execution time of your task and ISR execution profiles. For the moment, you can use some 'invented' execution times. This enables you to see how timing analysis is a simple step on from building your application.

Let's use the following times:

- Execution time for ISR *PrimaryISR.pButtonPress* is 1,000 processor cycles.
- Execution time for ISR *PrimaryISR.pMotor1Running* 1,500 processor cycles.
- Execution time for ISR *PrimaryISR.pTimer* is 2,000 processor cycles.
- Execution time for task *LampToggle* is 8,000 processor cycles. It toggles *Lamp3* on after 7,000 processor cycles.
- Execution time for task *Button1Response* is 20,000 processor cycles. It switches *Lamp1* on after 12,000 processor cycles and turns *Lamp2* off after 16,000 processor cycles.
- Execution time for task *MotorResponse* is 20,000 processor cycles. It switches *Lamp2* on after 10,000 processor cycles and turns *Lamp1* off after 14,000 processor cycles.
- Execution time for task *MotorStart* is 40,000 processor cycles. It applies power after 30,000 processor cycles.

Notice that you have specified execution time in processor cycles, rather than in the seconds or milliseconds that were used when specifying the real-world stimuli and their deadlines. RTA-OSEK knows that if the processor clock frequency doubles, the execution times halve, but that the stimuli and deadlines do not change. Take care to use appropriate units when entering time values.

Be aware that you can build some systems that RTA-OSEK cannot analyze. Fortunately, however, it is unlikely that you will come across this problem.

There are three simple rules governing analyzable systems:

- A task cannot activate a higher priority task. In fact, in a well-designed real-time system, tasks are normally activated by ISRs. The ISR reacts to a stimulus by activating one or more tasks to implement the responses. Sometimes such a 'response task' may pass work down to a lower priority worker-task, but it rarely needs to pass it to a higher priority task because OSEK resources are a more efficient way to perform part of the processing at 'high priority'.
- Tasks must have different priorities. Shared task priorities are often used in OSEK systems to ensure that certain tasks run in mutual exclusion. In most cases, you can actually set different task priorities for each task and use internal resources to enforce mutual exclusion. Incidentally, the RTA-OSEK OS implementation for systems that do not use shared priorities is more efficient than one that does use shared priorities, because it does not have to manage a FIFO queue for the tasks at each shared priority.
- The OSEK API call `Schedule()` cannot be used. The `Schedule()` call has the effect of releasing all internal task resources. This cannot be sensibly modeled by the timing analysis. You will find that it is very unlikely that you would need to use `Schedule()` in an application built with the RTA-OSEK GUI.

Important: You cannot analyze the response times, the extended tasks and any basic tasks of lower priority that the highest priority extended task.

Before RTA-OSEK will perform timing analysis on a system, you need to tell it that your application conforms to these rules.

- From the navigation bar, select the **Application** group and then select the **Optimizations** subgroup.
- Select the **No Upward Activation**, **Unique Task Priorities** and **Disallow Schedule()** checkboxes. You can also select **No RES_SCHEDULER**, because this standard OSEK resource is not used and will only cause unnecessary warning messages during analysis.

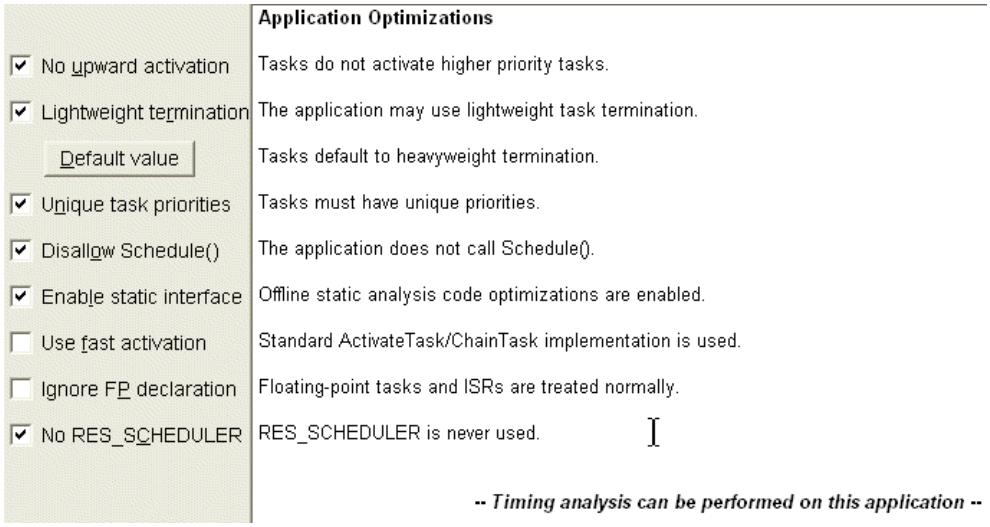

Application Optimizations
No upward activation
Tasks do not activate higher priority tasks.
Lightweight termination
The application may use lightweight task termination.
Default value
Tasks default to heavyweight termination.
Unique task priorities
Tasks must have unique priorities.
Disallow Schedule()
The application does not call Schedule().
Enable static interface
Offline static analysis code optimizations are enabled.
Use fast activation
Standard ActivateTask/ChainTask implementation is used.
Ignore FP declaration
Floating-point tasks and ISRs are treated normally.
No RES_SCHEDULER
RES_SCHEDULER is never used.
-- Timing analysis can be performed on this application --

Figure 3:62 - Selecting the Application Optimization Settings

Entering the Execution Times

To enter the execution times for the ISR *PrimaryISR*

- select the **ISRs** group from the *Planner* navigation bar.
- From the navigation bar, select the **Category 2 ISRs** subgroup.
- Select the ISR ***PrimaryISR*** and then select profile ***pButtonPress***. Click the **Execution Limits** button and set the execution limit to **1000 processor cycles**, as shown in Figure 3:63. If you wish, you can also specify the amount of stack that is used in this execution profile.

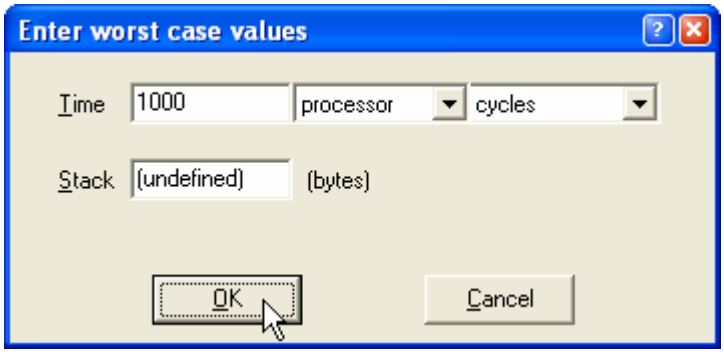


Figure 3:63 - Entering the Worst-Case Values for pButtonPress

- Select profile ***pMotor1Running*** and set its execution limit to ***1500 processor cycles***.
- Select profile ***pTimer*** and set its execution limit to ***2000 processor cycles***.

For the task *LampToggle*:

- From the **Tasks** group on the navigation bar, select the **Task Data** subgroup.
- Select task ***LampToggle*** and set its execution limit to ***8000 processor cycles***.
- Select the **Stimuli** group on the navigation bar and then select the **Stimuli** subgroup. Select the stimulus ***Lamp3Toggle*** from the drop down list.
- Click the **Implementation** button and set the execution time to ***7000 processor cycles***. This reflects the fact that the code in the task actually toggles the lamp some time before the end of the task.

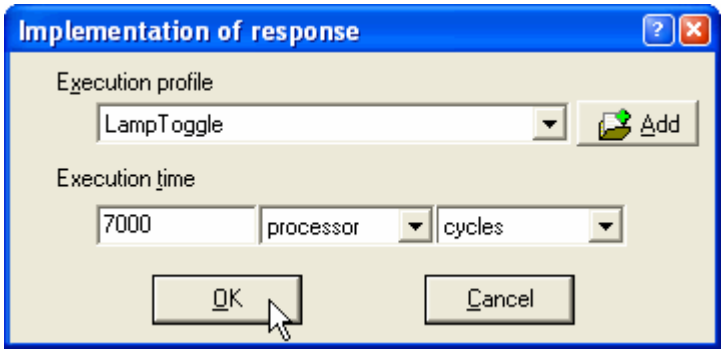


Figure 3:64 - Specifying the Execution Time for Lamp3Toggle

For task *Button1Response*:

- From the **Tasks** group on the navigation bar, select **Task Data** subgroup.
- Select task ***Button1Response*** and set its execution limit to **20000 processor cycles**.
- Select the **Stimuli** group on the navigation bar and then select the **Stimuli** subgroup. Select the stimulus ***Button1Press*** from the drop down list.
- Select response ***Lamp1On*** and set the implementation execution time to ***12000 processor cycles***.
- Select response ***Lamp2Off*** and set the implementation execution time to ***16000 processor cycles***.

For task *MotorResponse*:

- Select **Task Data** from the **Tasks** group of the navigation bar.
- Select task ***MotorResponse***. Set its execution limit to ***20000 processor cycles***.
- Select the **Stimuli** group on the navigation bar and then select the **Stimuli** subgroup. Select the stimulus ***Motor1Running*** from the drop down list.
- Select response ***Lamp2On*** and set the implementation execution time to ***10000 processor cycles***.
- Select response ***Lamp1Off*** and set the implementation execution time to ***14000 processor cycles***.

For task *MotorStart*:

- Select **Task Data** from the **Tasks** group of the navigation bar.
- Select task ***MotorStart*** and set its execution limit to ***40000 processor cycles***.
- Select the **Stimuli** group on the navigation bar and then select the **Stimuli** subgroup. Select the stimulus ***Button1Press*** from the drop down list.
- Select response ***Motor1On*** and set the implementation execution time to ***30000 processor cycles***.

Performing Schedulability Analysis

Now that you have got this far, you can perform timing analysis.

- From the **Analyze** group on the navigation bar, select the **Schedulability** subgroup.

The analysis results appear in the workspace.

Schedulability Analysis

Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.

Creating files

Analysis

*** Schedulability Analysis results ***

task MotorStart is schedulable.
 Calculated response time on MotorStart.default_profile for response Button1Press.Motor1On is 485700 cycles (60.7125 ms).
 Calculated response time on MotorStart.default_profile is 95700 cycles (11.9625 ms), with blocking 0 cycles.
task MotorResponse is schedulable.
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp2On is 46500 cycles (5.8125 ms).
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp1Off is 48900 cycles (6.1125 ms).
 Calculated response time on MotorResponse.default_profile is 52500 cycles (6.5625 ms), with blocking 0 cycles.
task Button1Response is schedulable.
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp1On is 27700 cycles (3.4625 ms).
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp2Off is 34100 cycles (4.2625 ms).
 Calculated response time on Button1Response.default_profile is 35700 cycles (4.4625 ms), with blocking 0 cycles.
task LampToggle is schedulable.
 Calculated response time on LampToggle.default_profile is 12500 cycles (1.5625 ms), with blocking 0 cycles.
interrupt primaryISR is schedulable.
 Calculated response time on primaryISR.pButtonPress is 6200 cycles (775 us), with blocking 2000 cycles (250 us), caused by IST primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pButtonPress is 1.
 Calculated response time on primaryISR.pMotor1Running is 4500 cycles (562.5 us), with blocking 2000 cycles (250 us), caused by IST primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pMotor1Running is 1.
 Calculated response time on primaryISR.pTimer is 4500 cycles (562.5 us), with blocking 0 cycles.
 Maximum buffer required on primaryISR.pTimer is 1.
 Maximum retriggers is 3.

The system is schedulable.

Figure 3:65 - Timing Analysis Text View

You can switch between the text and graphic views of the results by clicking on the **Text/Graphic** tabs at the bottom of the screen. (Right-click on the graphic to access the zoom in/out options.)

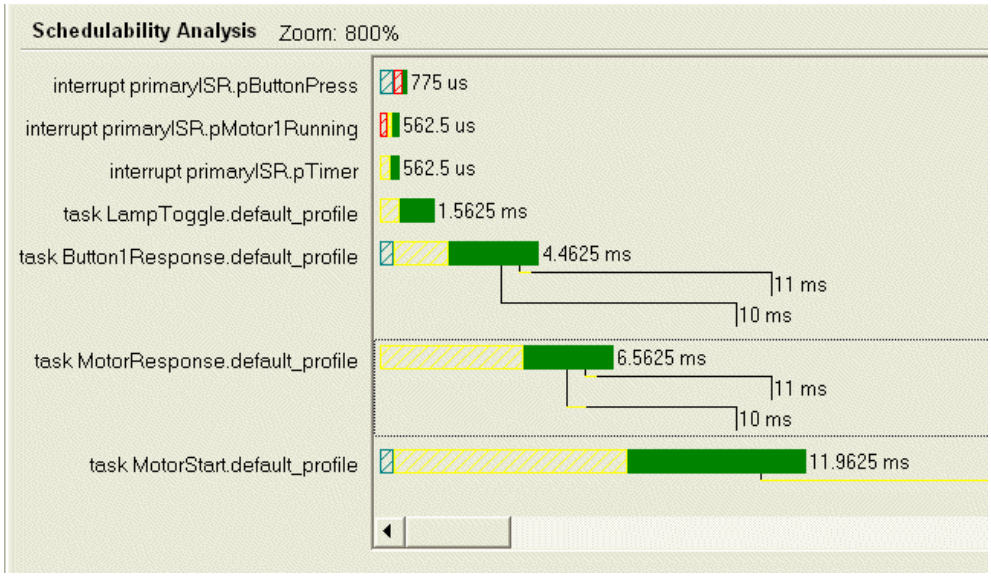


Figure 3:66 - Timing Analysis Graphical View

Figure 3:66 shows that this example system is schedulable. There are some points that you should note about the analysis:

- The profile *pButtonPress* of ISR *PrimaryISR* will always run and complete its execution within 775µs of *Button1* being pressed. This time includes up to 400µs for the button debounce delay, up to 250µs in which the profile can be blocked from starting by the execution of either *pMotor1Running* or *pTimer* and then the 125µs execution time of *pButtonPress* itself.
- The profile *pMotor1Running* of ISR *PrimaryISR* will always run and complete its execution within 562.5µs of the motor getting up to speed. This time includes up to 250µs in which the profile can be blocked from starting by the execution of *pTimer*, 125µs where *pButtonPress* can 'interfere' with it (if both interrupt sources are ready at the same time, *pButtonPress* takes precedence) and then the 187.5µs execution time of *pMotor1Running* itself.
- The profile *pTimer* of ISR *PrimaryISR* will always run and complete its execution within 562.5µs of the timer interrupt. This time includes up to 312.5µs in which the profile can be blocked from starting by the execution of *pButtonPress* and *pMotor1Running* and then the 250µs execution time of *pTimer* itself.
- Task *LampToggle* always terminates within 1.5625ms of the timer interrupt. This comprises up to 562.5µs interference from the ISR and 1ms execution time. You can see that the deadline of 4ms is met with 1.9375ms to spare, because the task issues the 'toggle' instruction by 1.5625ms and then there is 0.5ms response delay bringing the total response time to 2.0625ms.
- Task *Button1Response* completes within 4.4625ms and meets its deadlines.
- Task *MotorResponse* completes within 6.5625ms and meets its deadlines.

- Task *MotorStart* completes within 11.9625ms and meets its deadline.

You can try adjusting the execution times and deadlines to see the effect of the changes on the analysis results.

Interrupt Recognition and System Timings

In an actual system, the analysis must take account of time spent executing sections of OS code. For accurate analysis you should include the interrupt recognition and system timings.

Later on you will find out more about these and you will see how they should be calculated.

Performing Sensitivity Analysis

Sensitivity is used to determine the limits of schedulability of a system. Sensitivity analysis changes one parameter at a time and determines the maximum value it can take in a schedulable system.

To perform Sensitivity analysis

- From the **Analyze** group on the navigation bar, select the **Sensitivity** subgroup.

The analysis results appear in the workspace.

Sensitivity Analysis

Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.

Creating files

Analysis

*** Sensitivity Analysis results ***

--- Deadline sensitivity
In task MotorStart.default_profile, the deadline for response Button1Press.Motor1On can be met for execution time up to 40000 cycles (5 ms)
In task MotorResponse.default_profile, the deadline for response Motor1Running.Lamp2On can be met for execution time up to 20000 cycles (2.5 ms).
In task MotorResponse.default_profile, the deadline for response Motor1Running.Lamp1Off can be met for execution time up to 20000 cycles (2.5 ms).
In task Button1Response.default_profile, the deadline for response Button1Press.Lamp1On can be met for execution time up to 20000 cycles (2.5 ms).
In task Button1Response.default_profile, the deadline for response Button1Press.Lamp2Off can be met for execution time up to 20000 cycles (2.5 ms).

--- System sensitivity to execution and lock times
In task MotorStart.default_profile, the system can be schedulable for execution time up to 745100 cycles (93.1375 ms).
In task MotorResponse.default_profile, the system can be schedulable for execution time up to 725100 cycles (90.6375 ms).
In task Button1Response.default_profile, the system can be schedulable for execution time up to 53500 cycles (6.6875 ms).
In task LampToggle.default_profile, the system can be schedulable for execution time up to 41500 cycles (5.1875 ms).
In interrupt primaryISR.pButtonPress, the system can be schedulable for execution time up to 34500 cycles (4.3125 ms).
In interrupt primaryISR.pMotor1Running, the system can be schedulable for execution time up to 35000 cycles (4.375 ms).
In interrupt primaryISR.pTimer, the system can be schedulable for execution time up to 35500 cycles (4.4375 ms).

--- System sensitivity to clock speed
The system remains schedulable if processor clock speed is reduced to 58.13% of its current value.

Figure 3:67 - Sensitivity Analysis Text View

You can also view the results of the analysis in graphical format.

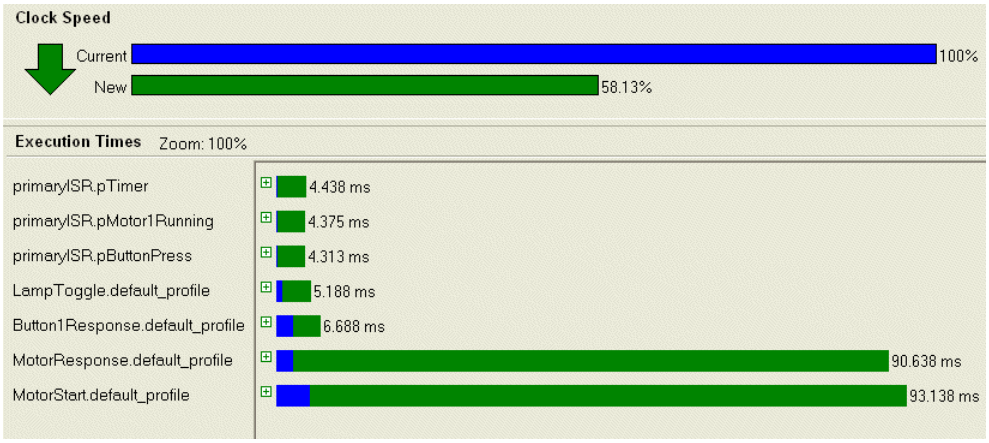


Figure 3:68 - Sensitivity Analysis Graphical View

You should note the following points:

- The code that turns *Lamp1* on and *Lamp2* off in task *Button1Response* can take as long as 20000 processor cycles to run and the system will still be schedulable. This means that the 'toggle' can be the last instruction in the task. In fact, at this clock speed, all of the 'critical execution' times in the system can be extended to the last instruction of the appropriate task.
- Task *Button1Response* can actually run for as long as 6.688ms.
- Task *MotorStart* can run for up to 93.138ms.
- Task *MotorResponse* can run for up to 90.638ms.
- Task *LampToggle* can run for up to 5.188ms.
- Execution profiles in *PrimaryISR* can run for up to 2.188ms, 2.25ms and 2.313ms.
- The CPU clock could be reduced to 58.13% of the declared value. This roughly halves the power needed by the processor.

Bear in mind that these are 'either/or' options. You should not expect to be able to apply all of these results and still have a schedulable system.

Calculating Best Task Priorities

Calculation of the "best task priorities" attempts to reduce the amount of task preemption (and hence stack usage) whilst keeping the system schedulable. It will suggest the ideal priority for each task, along with the internal resources that should be used.

- Select **Best Task Priorities** from the **Analyze** group of the navigation bar.

The results appear in the workspace.

```
*** Priority Allocation results ***

Task MotorStart is schedulable at priority level 2.
Task MotorResponse is schedulable at priority level 3.
Task Button1Response is schedulable at priority level 4.
Task LampToggle is schedulable at priority level 1.
Tasks LampToggle, MotorStart, MotorResponse, Button1Response must not preempt each other.

*** Schedulability Analysis results ***

task MotorStart is schedulable.
 Calculated response time on MotorStart.default_profile for response Button1Press.Motor1On is 485700 cycles (60.7125 ms).
 Calculated response time on MotorStart.default_profile is 95700 cycles (11.9625 ms), with blocking 8000 cycles (1 ms), caused by task
LampToggle.default_profile executing at its dispatch priority.
task MotorResponse is schedulable.
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp2On is 78500 cycles (9.8125 ms).
 Calculated response time on MotorResponse.default_profile for response Motor1Running.Lamp1Off is 80900 cycles (10.1125 ms).
 Calculated response time on MotorResponse.default_profile is 84500 cycles (10.5625 ms), with blocking 40000 cycles (5 ms), caused by
task MotorStart.default_profile executing at its dispatch priority.
task Button1Response is schedulable.
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp1On is 39700 cycles (4.9625 ms).
 Calculated response time on Button1Response.default_profile for response Button1Press.Lamp2Off is 46100 cycles (5.7625 ms).
 Calculated response time on Button1Response.default_profile is 47700 cycles (5.9625 ms), with blocking 20000 cycles (2.5 ms), caused
task MotorResponse.default_profile executing at its dispatch priority.
task LampToggle is schedulable.
 Calculated response time on LampToggle.default_profile is 92500 cycles (11.5625 ms), with blocking 0 cycles.
interrupt primaryISR is schedulable.
 Calculated response time on primaryISR.pButtonPress is 6200 cycles (775 us), with blocking 2000 cycles (250 us), caused by IST
primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pButtonPress is 1.
 Calculated response time on primaryISR.pMotor1Running is 4500 cycles (562.5 us), with blocking 2000 cycles (250 us), caused by IST
primaryISR.pTimer executing at interrupt priority 1.
 Maximum buffer required on primaryISR.pMotor1Running is 1.
 Calculated response time on primaryISR.pTimer is 4500 cycles (562.5 us), with blocking 0 cycles.
 Maximum buffer required on primaryISR.pTimer is 1.
 Maximum retriggers is 3.

The system is schedulable.
```

Figure 3:69 - Priority Analysis Text View

You can also see the results displayed graphically, as shown in Figure 3:70.

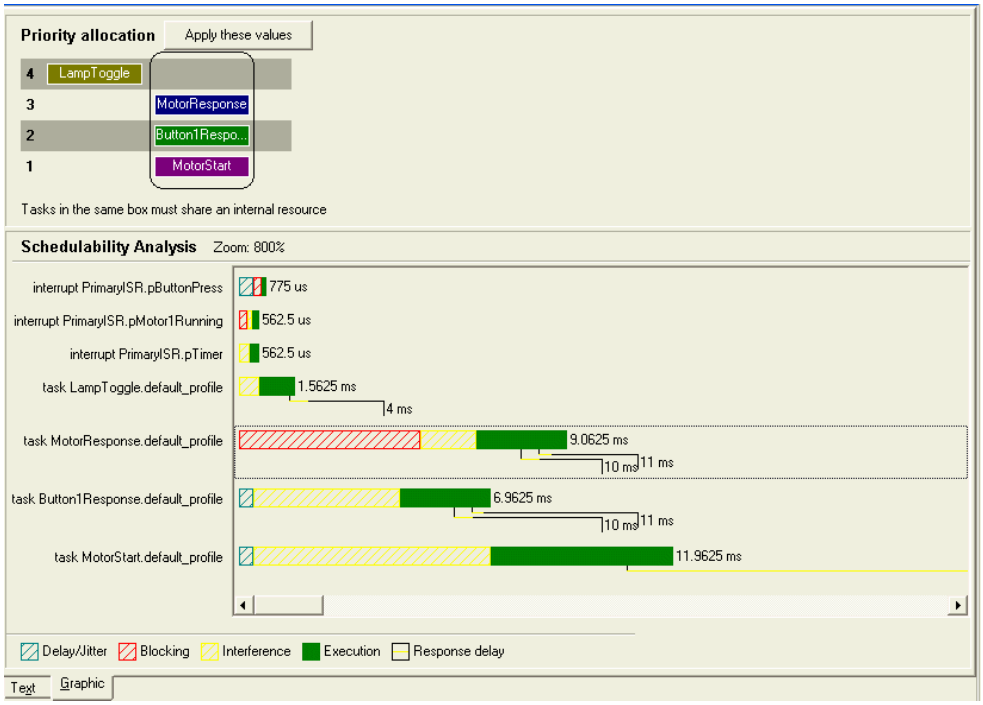


Figure 3:70 - Priority Analysis Graphical View

In this case, changing the task priorities and ensuring that Tasks *MotorStart*, *Button1Response* and *MotorResponse* do not preempt each other, can

reduced preemption. This can be achieved by assigning them to an internal resource.

The individual response times change when these settings are applied, but the system remains schedulable.

Calculating the CPU Clock Rate

CPU clock rate analysis attempts to reduce CPU clock rate whilst keeping the system schedulable. It will suggest the ideal priority for each task to achieve this clock rate.

- Select **CPU Clock Rate** from the **Analyze** group of the navigation bar.

The results appear in the workspace.

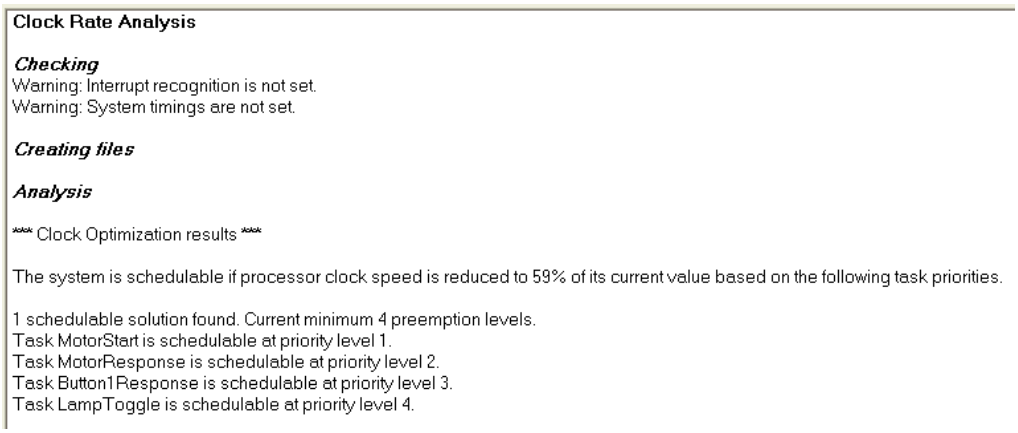

Clock Rate Analysis
Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.
Creating files
Analysis
*** Clock Optimization results ***
The system is schedulable if processor clock speed is reduced to 59% of its current value based on the following task priorities.
1 schedulable solution found. Current minimum 4 preemption levels.
Task MotorStart is schedulable at priority level 1.
Task MotorResponse is schedulable at priority level 2.
Task Button1Response is schedulable at priority level 3.
Task LampToggle is schedulable at priority level 4.

Figure 3:71 - CPU Clock Rate Analysis Text View

You can also see the results displayed graphically.

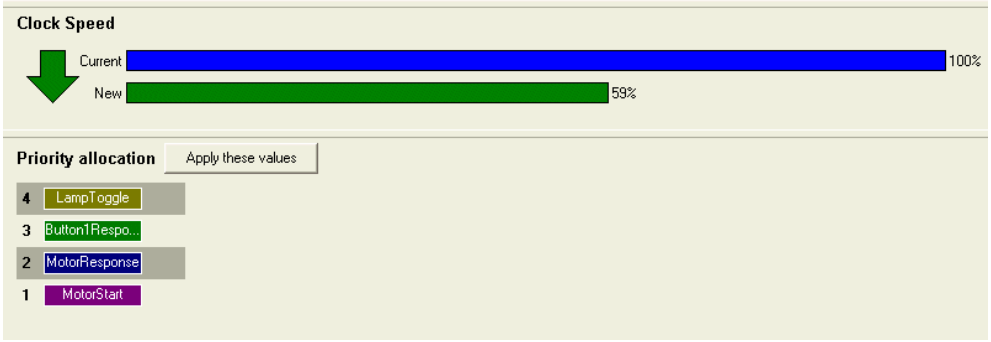


Figure 3:72 - CPU Clock Rate Analysis Graphical View

3.4 Completion of the Examples

You have now completed all of the stages required in these example applications. You have seen some sample specifications and the implementation of the requirements using the RTA-OSEK GUI.

You have learnt the basic skills, which include creating new applications and working with stimuli, responses, tasks and ISRs. You have also seen how to write code for applications.

You have learnt about functional testing and building an application. You have also seen a summary of the analysis options that are available.

Now you are encouraged to read the remainder of this guide to find out more about the extensive features available in RTA-OSEK.

3.5 RTA-OSEK *Builder*

The *Builder* contains two parts:

- A basic entry method for constructing an application, suitable for those already familiar with OSEK concepts. This is described in Section 3.5.1
- Application build (setting options and actually building the application). This is described in Section 3.5.2

In the following sections, we will go through the options available in the *Builder*.

3.5.1 Basic Data Entry

If you are already familiar with OSEK concepts, you may find that the extra features in the RTA-OSEK *Planner* are more than you need. The *Builder* provides another way to create and modify your application through a navigable grid-based interface that only shows standard OSEK features.

Application data is entered using the 'Basic Data Entry' view. Each class of OSEK object has its own tab – use the **Add** and **Remove** buttons to create and delete OSEK objects.

The tabbed data grids are selected to view all of the details for the individual OSEK objects. The **Tasks** tab is shown in Figure 3:73.

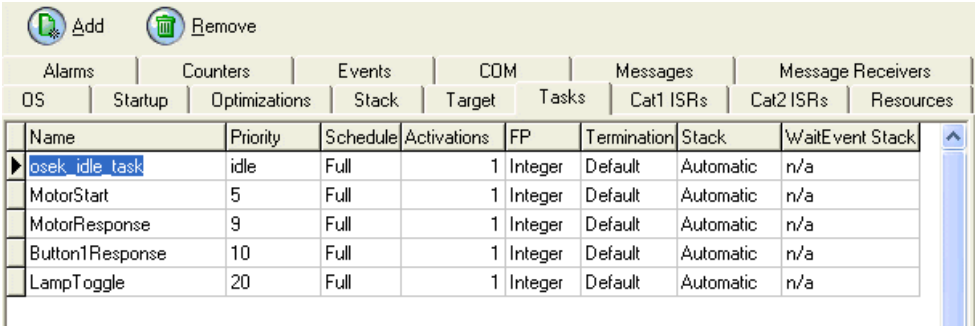


Figure 3:73 - Entering Tasks using the OSEK Wizard

3.5.2 Building an application

If you have successfully completed all of the steps in creating an application, you can now start the build process. The build process involves:

- Compiling the task and ISR C files.
- Compiling the RTA-OSEK generated C file `osekdefs.c`. This file contains data describing the RTA-OSEK component objects used in your application.
- Assembling the RTA-OSEK generated assembler file `osgen`. (The file extension is target-specific). This file contains data describing the low-level RTA-OSEK OS data.
- Compiling any additional supporting C files, such as target-specific files used to implement responses and initialize the hardware.
- Linking the resulting files with the RTA-OSEK OS API library, the compiler's run-time library and any run-time startup code.

There are two different ways to build your application. You can build a system manually or use a custom build script:

- A manual build refers to building the application outside of the RTA-OSEK GUI. The manual build process is outlined in Section 3.5.4.
- A Custom Build script can be constructed if you want to complete the build process entirely within the RTA-OSEK GUI. The Custom Build process is described in Section 3.5.5

3.5.3 Build Checks

Clicking on the Build Checks button will check that the system has been specified completely enough to build. This step does not compile or check any code; it simply confirms that required objects have been defined (whether in *Planner*, or in the basic data entry view).

If all required objects have been defined, each check will be marked with 'OK'.

Refer to the binding manual for your target for a list of required objects.

3.5.4 Create Files

This option generates a number of source files (C source and header files, as well as assembler source files) derived from the application specification. These generated files need to be incorporated into an external build process (perhaps using `make` or similar).

The process also creates an ORTI debugger file (if a debugger has been selected in the target configuration) and an RTA-TRACE description file (if tracing has been enabled).

If there are no build errors, the RTA-OSEK GUI will create and list the necessary files. An example is shown in Figure 3:74.

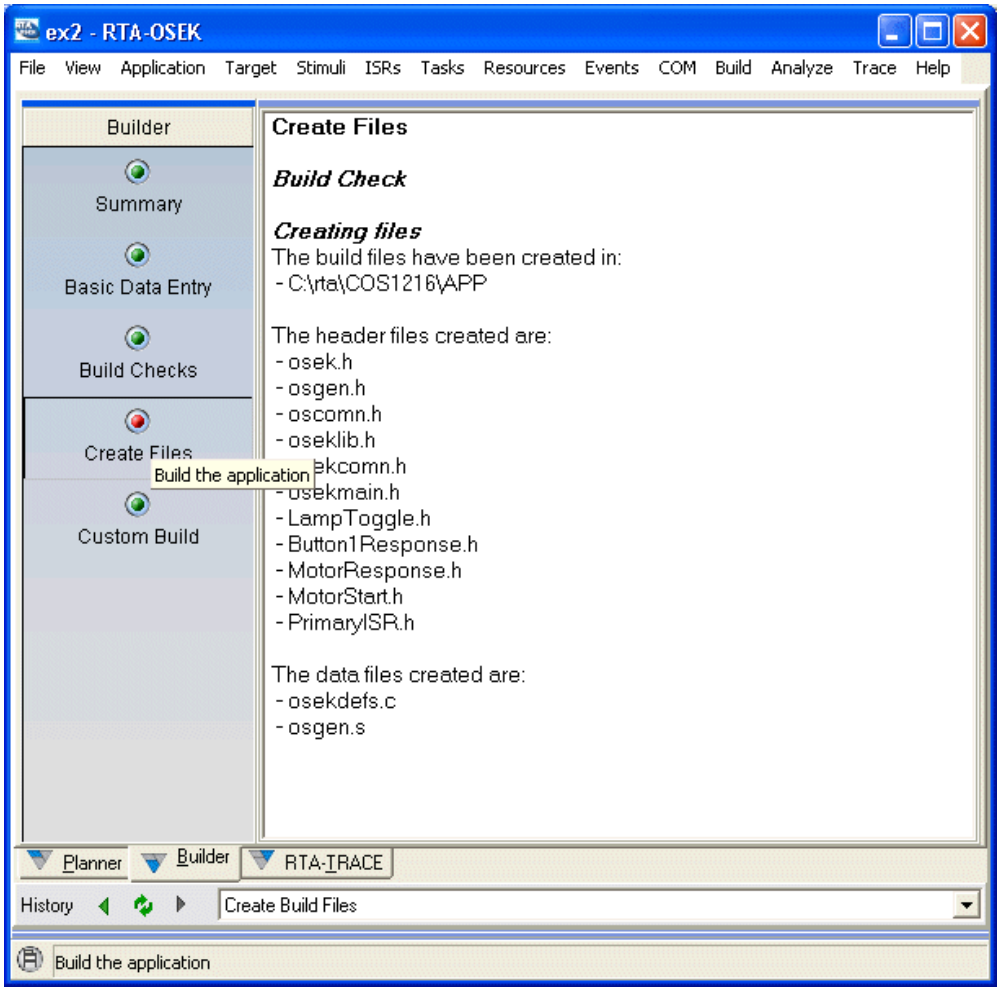


Figure 3:74 – Creating RTA-OSEK Files

Your application must be linked with the correct RTA-OSEK component library. Remember that the library used depends on the build status (Standard, Timing or Extended). The name of the library and its location are shown in the application implementation notes.

Important: When you are compiling and assembling your application, use the **rtkbuild.bat** file as a guide. The RTA-OSEK GUI can generate this file by using the **Create 'rtkbuild.bat'** button from the **'Custom Build'** view. In particular, take great care if you use compiler or assembler options other than those specified for `osekdefs.c` and `osgen.s`.

Once the files have been created, your external build tools (i.e. `make`) can be used to compile, assemble, and link the source into an executable.

3.5.5 Custom Build

The custom build process revolves around the build script `rtkbuild.bat` – this batch file will perform the compilation/assembly of your tasks, ISRs, `osekdefs` and `osgen` files.

`rtkbuild.bat` may have been generated already if the **Create Templates** button was used prior to implementation. Alternatively, if the file does not exist, it can be generated by using the **Create 'rtkbuild.bat'** button.

Note: If the system configuration has changed between the initial generation of `rtkbuild.bat` and the build itself, `rtkbuild.bat` will need to be regenerated.

The only steps you need to add are compile/assembly for any other files that are needed and then link/locate the object modules. This is configured using the **Custom Build Options** dialog, displayed by clicking the **Configure** button – See Section 3.5.6 for a full description.

Once the build script has been finalized:

- Save the application.
- Click the **Build Now** button.

The RTA-OSEK GUI checks for errors in the system description. If it detects a mistake, it will generate an error message and stop. If it discovers something that is unusual, but that may be correct, it will issue a warning and continue.

The RTA-OSEK GUI then runs the script. You will see the tool output displayed in the RTA-OSEK GUI window as the script is executed. If the script completes successfully, a new executable file will have been created, ready for testing.

3.5.6 Custom Build Options

The **Custom Build Options** dialog box is displayed by clicking the **Configure** button. This dialog allows the build script to be edited, environment variables to be set, and custom buttons to be defined.

Build Script

By default, the build script simply contains '`call rtkbuild.bat`'. In order to build the other components of the application, the script needs to be extended.

Let's assume that you have put the target-specific initialization and response implementation code in a file called `target.c`. You must add a line like this to the build script:

```
%CC% %COPTS% target.c
```

Note here that `cc` and `copts` are environment variables that have already been set up in `rtkbuild.bat`. You may choose to name the compiler and options more explicitly. You can also add a `-debug` option to the compiler

command-line. This is achieved by adding an environment variable `APP_COPT`.

The link/locate stage tends to be more target-specific. An example version, in a single line of code, is shown below.

```
%lnk% -v -l%RTA_LIB% -l%CBASE%\lib -m$(NAME).map -
otemp.out link.lkf $(RTKOBJECTS) target.$(OBJEXT)
$(RTKLIB) crtsi.$(LIBEXT) libm.$(LIBEXT)
libi.$(LIBEXT)
```

The **Custom Build Options** dialog will look something like Figure 3:75.

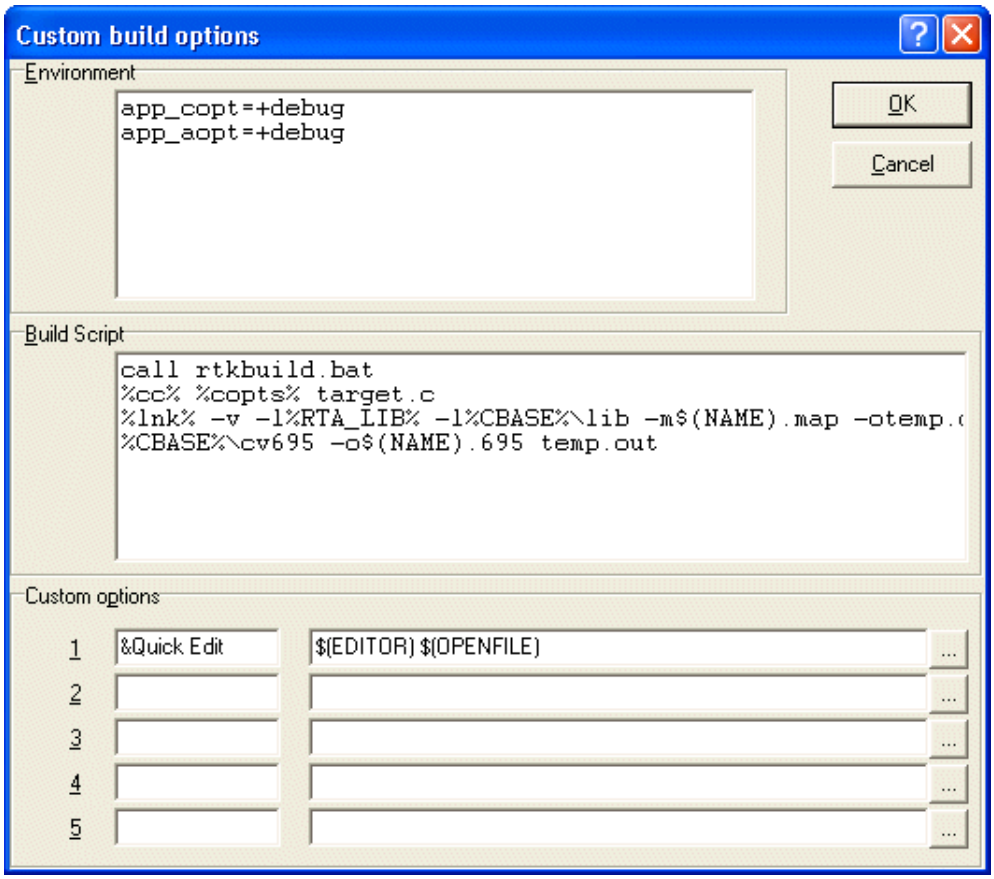


Figure 3:75 - Creating a Custom Build Script

The words starting with '`%`' refer to environment variables set up via `rtkbuild.bat`.

A full list of the environment variables that are normally available can be seen in the following table.

Name	Content	Definition
CBASE	The base location of your compiler and tools.	Toolinit.bat
CC	The fully pathed name of your compiler.	Toolinit.bat
AS	The fully pathed name of your assembler.	Toolinit.bat
LNK	The fully pathed name of your linker.	Toolinit.bat
AR	The fully pathed name of your librarian.	Toolinit.bat
CBASE_INC	The path name to your compiler's include files.	Toolinit.bat
COPTS	Default options for compiling C source code file in rtkbuild.bat. You can add options to this by declaring an extra environment variable APP_COPT in the environment section of the custom build setup. e.g. APP_COPT=-debug	rtkbuild.bat
AOPTS	Default options for assembling files in rtkbuild.bat. You can add options to this by declaring an extra environment variable APP_AOPT in the environment section of the custom build setup.	rtkbuild.bat

The words with '$()' such as $(NAME), used above, refer to RTA-OSEK GUI macro variables. These can be used in any of the custom build configuration entries. They are expanded to the appropriate values during the final build.

All of the macro variables are listed in the following table

Name	Content
$(ASMEXT)	The extension used for assembler files with this compiler toolchain, for example asm.
$(CONFORMANCE)	This macro indicates certain aspects of the build – essential for building libraries. '1' – no shared priorities in the system; '2C' – at least one task has multiple activations, but no shared priorities; '2F' – at least two tasks have the same priority; 'e' – used as a qualifier to the above three indicators, this shows whether events are used (i.e. an ECC system). In summary, there are six permutations: '1','1e','2C','2Ce','2F', and '2Fe'
$(DIR)	The home directory of the application. This is the directory in which the OIL file is stored and into which the build files are generated.
$(EDITOR)	The name of the default editor used by the RTA-OSEK GUI. By default, this is the Windows Notepad application, but you can specify your own editor via the RTA-OSEK GUI System Options (accessed by selecting the Options from the File menu).
$(LIBEXT)	The extension used for library files with this compiler toolchain, for example lib.
$(LIBTYPE)	'S' 'T' or 'E' (corresponding to 'Standard', 'Timing' or 'Extended').
$(NAME)	The application name. For example, 'UserApp'.
$(OBJEXT)	The extension used for object files with this compiler toolchain, for example 'obj'.
$(OPENFILE)	Causes an 'Open File' dialog to be shown and returns the name of the file selected. The 'Quick Edit' button has been created using

Name	Content
	the line `$(EDITOR) $(OPENFILE)`.
`$(OS_STATUS)`	'Standard', 'Timing' or 'Extended'.
`$(RTABASE)`	Directory of `RTAINIT.BAT`. The installation directory for RTA tools and targets.
`$(RTKOBJECTS)`	A list of all the task and ISR object files that are created within `rtkbuild.bat`. Space separated.
`$(RTKOBJECTS_C)`	A list of all the task and ISR object files that are created within `rtkbuild.bat`. Comma separated.
`$(RTKLIB)`	The name of the RTA-OSEK component library to which the application must link. This differs, for example, between Standard and Extended builds.
`$(TARGET)`	The target name, for example, 'HC12/COSMIC 16 task'.
`$(TGTBASE)`	Directory of `TOOLINIT.BAT`. The installation directory for the current target.
`$(VARIANT)`	The target variant. For example, 'Star12'. This is commonly passed as a command-line 'define' when compiling target-specific code, so that appropriate settings can be selected for different chip variants.

Environment

The **Custom Build** dialog **Environment** section is used to declare any environment variables you need to use as part of the custom build process. You can use RTA-OSEK GUI macros when defining these. For example:

```
DEST=$(DIR)
```

Custom Options

The **Custom Options** section allows you to create up to five user-defined buttons for custom builds. By default the first button is set up as a 'Quick

Edit'. You can configure these buttons to launch any external programs, such as a source-code control system or debugger.

3.6 Other Implementation Details

When writing your own applications, you should consider the implementation details in this section.

3.6.1 Namespace

The RTA-OSEK component has a defined **namespace**. Names cannot be created that conflict with existing names or names used internally. Internal names used by the RTA-OSEK component generally begin with the prefixes 'os' or 'OS' or '_os' or '_OS'. Other internal names include the tokens used in the enhanced OIL grammar. Follow these simple rules:

- Do not pick names beginning with 'os' or 'OS' or '_os' or '_OS'. These are all reserved for the RTA-OSEK component.
- Do not choose an object module or file name beginning with 'os'.

3.6.2 Reentrancy

All calls to the RTA-OSEK component are reentrant where necessary. Special protection to prevent reentry is not required. The C libraries provided by your compiler supplier, however, may not be reentrant (most C libraries are not).

Floating-point support presents a common reentrancy problem with C libraries. The floating-point problem is not always obvious, since the compiler can insert calls to floating-point libraries silently. With the RTA-OSEK component, floating-point can be used safely in tasks and ISRs by specifying that the object uses floating-point.

Portability: On some targets, reentrancy problems can occur when functions return structures. If this is the case, protection against reentrancy must be performed before and after the call is made to a function that returns a structure.

Important: It is your responsibility to prevent reentry to a non-reentrant function. This is usually implemented by disabling interrupts or by using RTA-OSEK component 'resources'. In fact, a C library of non-reentrant functions may contain 'hooks' where RTA-OSEK component resource get and release calls can be inserted to protect against reentry.

Any reentrant function in a system running the RTA-OSEK component need only be **serially reentrant**, as opposed to **fully reentrant**. A serially reentrant function is one where it is acceptable to switch from a thread of control currently executing the function to a thread of control that is not yet executing the function, but will do so later.

Some compilers can generate different code for reentrant and nonreentrant functions. For example, nonreentrant functions can use static data overlaying techniques for parameters and local variables. A reentrant version will use the stack.

RTA-OSEK provides the `OS_REENTRANT` and `OS_NONREENTRANT` macros to ensure generation of the appropriate code. For compilers where there is no difference in code generated, these macros have no effect. However, it is recommended that they be used for portability. These macros are described in detail in the *RTA-OSEK Reference Guide*.

The RTA-OSEK component does not normally use any functions from the standard C library, but may need to use some target-specific code from the C compiler library. The RTA-OSEK component does not make floating-point calculations itself. Refer to the *RTA-OSEK Binding Manuals* for information on the requirements of the RTA-OSEK component on each particular target.

The Development Process Summary

- RTA-OSEK provides facilities for the specification, design, implementation, building and analysis of hard real-time systems.
- Applications are modeled as stimulus/response relationships. The associated performance constraints are expressed as deadlines on responses.
- The design and implementation determines how stimuli are captured and how the responses are generated in terms of OSEK OS objects.
- You can build entire applications using a custom or manual build in the simple development environment interface.
- When execution times for tasks and ISRs in your application are determined, timing analysis can then be performed on the stimulus/response model to show that all performance constraints are satisfied at run-time.

4 Tasks

A system that has to perform a number of different activities at the same time is known as **concurrent**. These activities may have some software component, so the programs that provide them must execute concurrently. The programs will have to cooperate whenever necessary, for example, when they need to share data.

Each concurrent activity in a real-time system is represented by a **task**. The majority of the application code exists within tasks.

If you have a number of tasks that must be executed at the same time, you will need to provide a means to allow concurrency. One way for you to do this is to have a separate processor for each task. You could use a parallel computer, but this solution is too expensive for many applications.

A much more cost effective way for you to achieve concurrent behavior is to run one task at a time on a single processor. You can then switch between tasks, so that they appear to be executing at the same time.

4.1 Task Switching

A **scheduler** is used to perform task switching. It does this by implementing a **scheduling policy.** The policy dictates when one task should (temporarily) stop executing and another task should start.

The OSEK operating system specifies a scheduler that uses a **fixed priority** scheduling policy.

Under this policy, each task is assigned a fixed priority. The scheduler will always run the highest priority task that is ready to run. If a task is *running* and a higher priority task is made *ready* to run. The higher priority task will **pre-empt** the lower priority task. When the higher priority task has finished executing, the lower priority task is **resumed** at the point of pre-emption.

You can see an illustration of this in Figure 4:1.

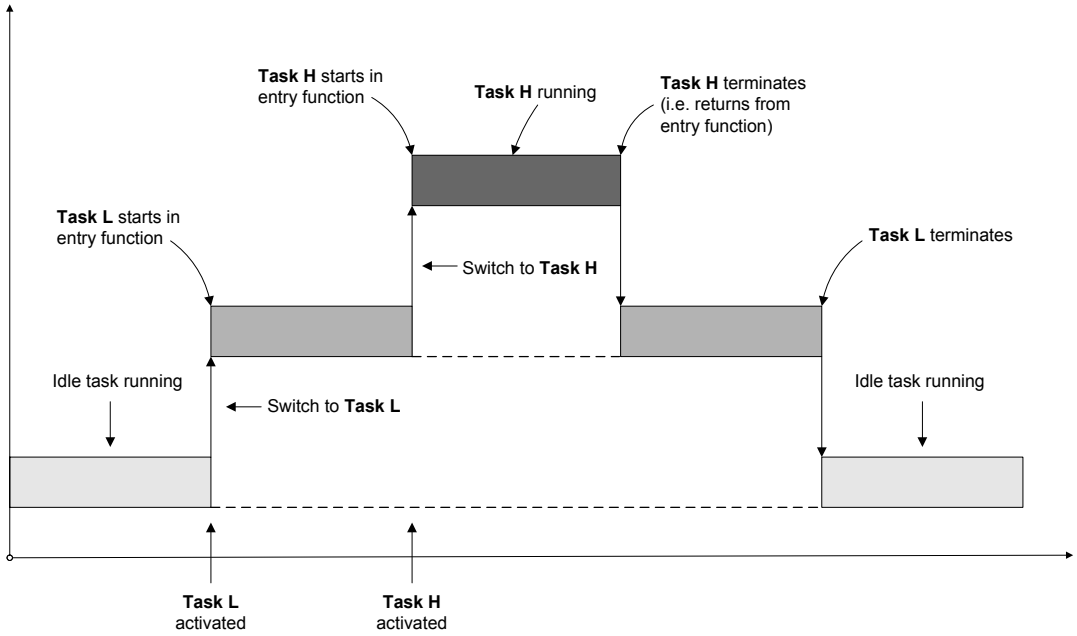

Task H starts in entry function
Task H running
Task H terminates (i.e. returns from entry function)
Task L starts in entry function
Switch to Task H
Task L terminates
Idle task running
Switch to Task L
Idle task running
Task L activated
Task H activated

Figure 4:1 - Example Execution of Tasks

In Figure 4:1 you can see that, initially, the idle task is *running* (you will learn about the idle task in Section 4.7). At some point a low priority task, `L`, is activated. A task switch takes place and `L` starts executing from the start of its entry function.

Later, a higher priority task, `H`, is activated and again a task switch takes place. `H` starts executing from the beginning of its entry function.

`H` then terminates and `L` resumes execution from the point it was preempted. `L` eventually terminates. Finally, the idle task resumes execution from the point at which it was preempted.

4.2 Basic and Extended Tasks

OSEK operating systems categorize tasks into two classes. These classes are called **basic tasks** and **extended tasks**. The task class specifies the states, in the operating system state model, that are valid for a particular task.

Within each class, there are two levels. These levels are called type 1 and type 2. You'll learn about these later in this chapter.

4.2.1 Basic Tasks

Basic tasks are **single-shot** tasks. This means that a task is made *ready* and then starts executing from its entry point. During execution it may be preempted by other higher priority tasks, but it will continue to run (whenever there are no higher priority *ready* tasks) until termination. It can be made *ready* again later and the task can execute again.

Basic Task States

Basic tasks can exist in the following states:

- Ready.
- Running.
- Suspended.

The default state for a task is *suspended*. A task is moved into the *ready* state either by an explicit activation API call or by some other method that will cause activation. The state transition diagram for basic tasks is shown in Figure 4:2.

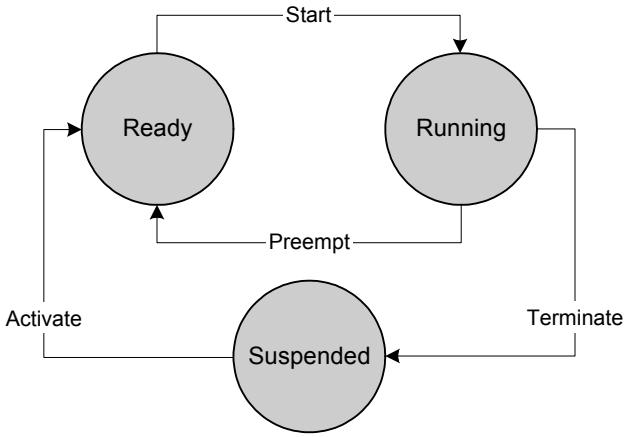


Figure 4:2 - The State Transition Behavior for Basic Tasks

Looking at Figure 4:2, you can see that when RTA-OSEK Component chooses to run a task it moves from the *ready* state to the *running* state. The execution of the task starts from the task entry point.

If a higher priority task becomes *ready* to run, the currently executing task is preempted and is moved from the *running* state into the *ready* state. Only one task can be in the *running* state at any one time.

A task returns to the *suspended* state by **terminating**.

Important: Basic tasks cannot wait for a specific event or delay for a certain time (other than busy waiting). In Figure 4:2, you can see that the only way of a task becoming *suspended* is by terminating.

Type 1 and Type 2 Basic Tasks (BCC1 and BCC2)

You saw earlier that within each conformance class there are two levels. The Basic Conformance Class (BCC) has type 1 and type 2 tasks.

- Type 1 basic tasks (**BCC1**).
  These are single-shot tasks. They have a unique task priority and they cannot be activated unless they are currently in the *suspended* state.
- Type 2 basic tasks (**BCC2**).
  These are single-shot tasks. They can share a priority with another task and they can have **multiple activations**. Multiple activation means that a task can be activated, up to a specified number of times,

whilst it is in the *ready* or *running* state. You'll find out more about this in Section 4.8.

If two or more tasks have the same priority, each task at the shared priority will run in **mutual exclusion**. This means that if one task is *running*, the other tasks sharing the same priority cannot pre-empt it. Their activations are queued in a FIFO manner. As a result of this, you will not be able to perform timing analysis on the system.

Important: To make RTA-OSEK Component more efficient, you should assign unique task priorities and use internal resources to enforce mutual exclusion. If you do this timing analysis will also be possible.

The RTA-OSEK GUI allows you to set the maximum number of times that task activations can be queued. RTA-OSEK Component will ensure that the task executes once for each of the activations that is recorded, up to limit that you set. Where more than one task shares the same priority, the tasks are run strictly in the order that they were activated.

4.2.2 Extended Tasks

Extended tasks usually exist in infinite loops. Once they are *running*, they do not normally terminate. They can 'sleep' in a *waiting* state, pending the outcome of an event.

Extended Task States

Extended tasks can exist in the same three states as basic tasks:

- Ready.
- Running.
- Suspended.

In addition to these, they can also exist in an extra state:

- Waiting.

The state diagram, in Figure 4:3, shows the four states for an extended task in an OSEK operating system. You will notice that the extended task behavior in the *ready*, *running* and *suspended* states is identical to behavior of basic tasks (in Figure 4:2).

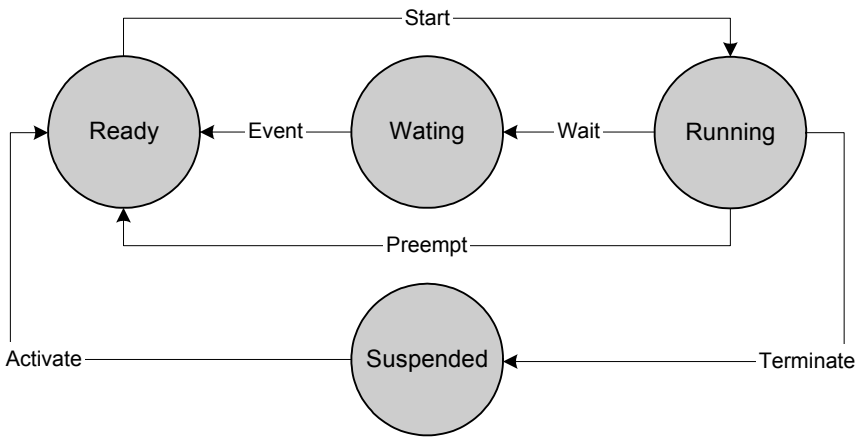


Figure 4:3 - The State Transition Behavior for Extended Tasks

An extended task moves from the *running* to the *waiting* state when it voluntarily suspends itself by *waiting* on an event.

An event is simply a system object that is used to provide an indicator for a system event. Examples of events include data becoming ready for use or sensor values being read. When an event is set, the task is moved from the *waiting* to the *ready* state.

If an extended task is *waiting* on an event, then tasks of lower priority are allowed to run.

Type 1 and Type 2 Extended Tasks (ECC1 and ECC2)

Again, as you saw earlier, each conformance class contains two levels. The Extended Conformance Class (ECC) has type 1 and type 2 tasks.

- Type 1 extended tasks (**ECC1**).
  These tasks can wait for events and have unique task priorities. So, an ECC1 task is like a BCC1, but it can wait on events.
- Type 2 extended tasks (**ECC2**).
  These tasks can wait for events and can have the same priority as other tasks. Where more than one task shares the same priority, the tasks are run strictly in the order they were activated. Note that, unlike type 2 basic tasks, type 2 extended tasks cannot use multiple activation.

Important: Extended tasks are not amenable to timing analysis. RTA-OSEK will limit any analysis to basic tasks and ISRs that are of a higher priority than any extended task. This means that you should make sure that the hard real-time aspects of your system are of a higher priority than the highest priority extended task.

4.3 Simple Task Configuration

Unlike other real-time operating systems that you might have seen, the tasks in OSEK (and, therefore, in RTA-OSEK) are defined statically. This technique is used because it saves RAM and execution time.

Tasks cannot be created or destroyed dynamically. Most of the information about a task can be calculated offline, allowing it to be stored in ROM.

When you configure your task properties, you will use the RTA-OSEK GUI. Look at the example in Figure 4:4 to see how a task has been constructed.

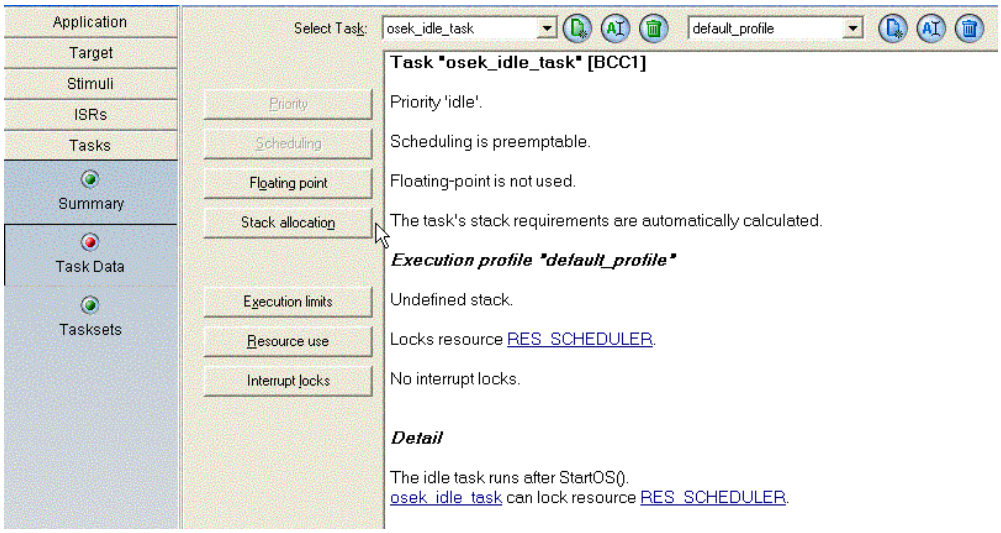


Figure 4:4 - Configuring a Task in the RTA-OSEK GUI

At the simplest level a task has the following attributes:

- A task name.
  The name is used to refer to, or provide a **handle** to, C code that you will write to implement the task functionality.
- A task priority.
  The priority is used by the scheduler to determine when the task runs. Priorities cannot be changed dynamically.

Portability: The number of tasks that can be defined is fixed for each target (it is usually 16 or 32, depending on the target processor). The supplied *RTA-OSEK Binding Manual* for your specific target will contain further information.

If you have any extended tasks in your application you will also need to specify the amount of stack space that they use. In the RTA-OSEK GUI, you should enter the stack allocation to specify the worst-case stack size. When an extended task waits on an event, it must be moved off the stack temporarily.

For basic tasks you can allow RTA-OSEK to automatically calculate the stack requirements. This information is needed because RTA-OSEK Component runs all tasks on a single stack.

RTA-OSEK uses the stack information that you specify in stack allocation to determine the size of the memory area it needs to reserve for this move.

You can specify a default stack size in the Application Defaults for the RTA-OSEK GUI. In Figure 4:5, the default stack size has been set to 100 bytes.

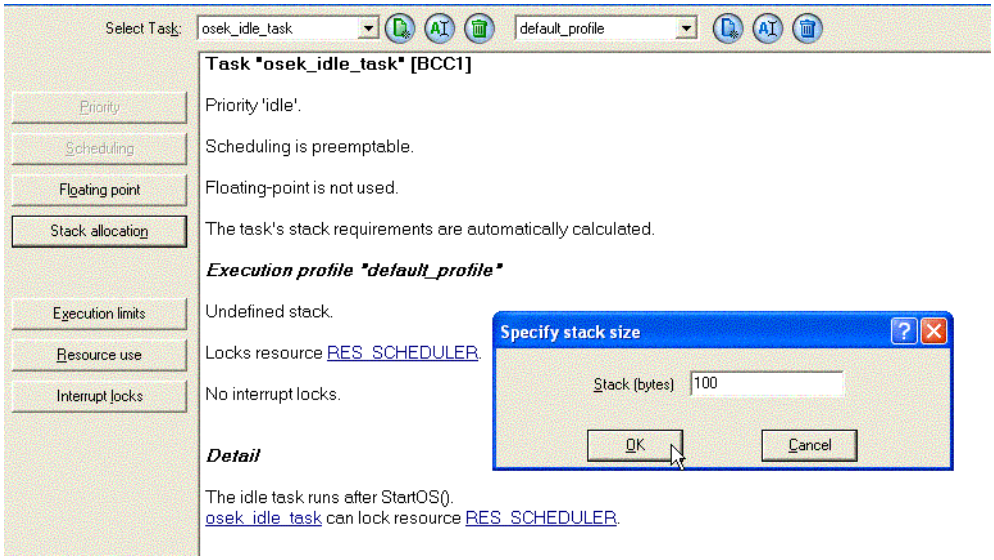


Figure 4:5 - Specifying a Default Stack Size

The default stack size will be used for each task, unless you specify a stack allocation or stack usage information in the task profile.

4.4 Task Entry

Tasks are similar to C functions that implement some form of system functionality when they are called by RTA-OSEK Component. Just like any other C function, they can be invoked from the task.

When a task starts *running*, execution begins at the task **entry function**. The task entry function is written using the C syntax in Code Example 4:1.

```
TASK(task_identifier)  { … } /* Task entry
function. */
```

Code Example 4:1 - A Task Entry Function

Remember that basic tasks are single-shot. This means that they execute from their fixed task entry point and terminate when completed.

Code Example 4:2 shows the code for a basic task called `Task1`.

```
#include "Task1.h" /* Header file generated RTA-
OSEK. */

TASK(Task1) {

  do_something();
  TerminateTask(); /* Task must finish with
                    * TerminateTask() or
                    * equivalent. */
}
```

Code Example 4:2 - A Basic Task

Now, compare the example in Code Example 4:2 with Code Example 4:3. Code Example 4:3 shows that extended tasks need not necessarily terminate and can remain in a loop waiting for events.

```
#include "Task2.h"    /* Header file generated
                       * by RTA-OSEK. */
TASK(Task2) {

  do_something();

  while (WaitEvent(Event)==E_OK) {
    do_something_else();
    ClearEvent(Event);
  }

  /* Task never terminates. */
}
```

Code Example 4:3 - Extended Task Waiting for Events

Important: You do not need to provide any C function prototypes for task entry functions. These are provided in the header file generated by RTA-OSEK. The appropriate file for each task should be included because it contains declarations that are specific to the named task.

4.5 Task Termination

4.5.1 API Calls for Task Termination

The OSEK standard defines two API calls for task termination. One of these must be used to terminate any task. These API calls are:

- `TerminateTask()`
- `ChainTask(TaskID)`

In addition to this, RTA-OSEK Component also offers an extra method of task termination:

- `ChainTaskset(TasksetID)`

These three API calls are all known as the **Terminating API**. When a task has finished, it must call the appropriate terminating API. This ensures that RTA-OSEK Component can correctly schedule the next task that is *ready* to run.

In addition to this, `ChainTask(TaskID)` also causes the named task to be placed into the *ready* state and `ChainTaskset(TasksetID)` puts all the tasks in the taskset into the *ready* state.

4.5.2 Heavyweight and Lightweight Termination

The OSEK operating system standard allows the terminating API to be called by a task at any point, including within a deeply nested set of function calls.

In many applications this imposes an unnecessary overhead on the system. For this reason, the RTA-OSEK GUI allows you to specify that you will only call the terminating API from the task entry function. If you specify this, RTA-OSEK Component does not impose any overhead for managing deeply nested termination on the task.

In Code Example 4:4, the task entry function makes nested calls to other functions.

```
#include "Task1.h"      /* Header file generated by
                         * RTA-OSEK. */

Task(Task1) {

  /* Make a nested function call. */
  Function1();

  TerminateTask();  /* Terminate the task. Normally
                     * the stack is tidied up by
                     * the function return. */
}
```

```
#include "osek.h"       /* Header file generated by
                         * RTA-OSEK. */
void Function1(void) {
  ...
  Function2();
  ...
}

void Function2(void) {
  If (SomeCondition) {
    TerminateTask();    /* Terminate current calling
                         * task. The OS must tidy up
                         * the stack. */
  }
}
```

Code Example 4:4 - Terminating a Task

From Code Example 4:4, you can see that when `Task1` runs, it calls `Function1()`. `Function1()` then calls `Function2()`. In `Function2()` there is some code that can terminate the calling task (in this example, this is `Task1`).

RTA-OSEK uses the following terms:

- **Heavyweight termination**.
  Describes the situation where the terminating API can be called from within a nested function.
- **Lightweight termination**.
  Used to describe cases where the terminating APIs are called only from the task entry function.

With heavyweight tasks, RTA-OSEK Component must store information that allows it to clear the stack when the task terminates somewhere other than the entry function. This is normally done using a `setjmp`/`longjmp` pair.

If tasks are only terminated in the entry function, however, this information does not need to be stored. Specifying that a task is lightweight tells RTA-OSEK not to generate code to save this information and, as a result, you will save stack space.

In the RTA-OSEK GUI, you can set the default termination type to be either lightweight or heavyweight. When you configure an individual task, you can then set the termination type to the use the application default.

It is recommended that the default termination type should be set to 'lightweight' and, wherever possible, task termination should be set to 'default'. Figure 4:6 shows how the default termination type is set.

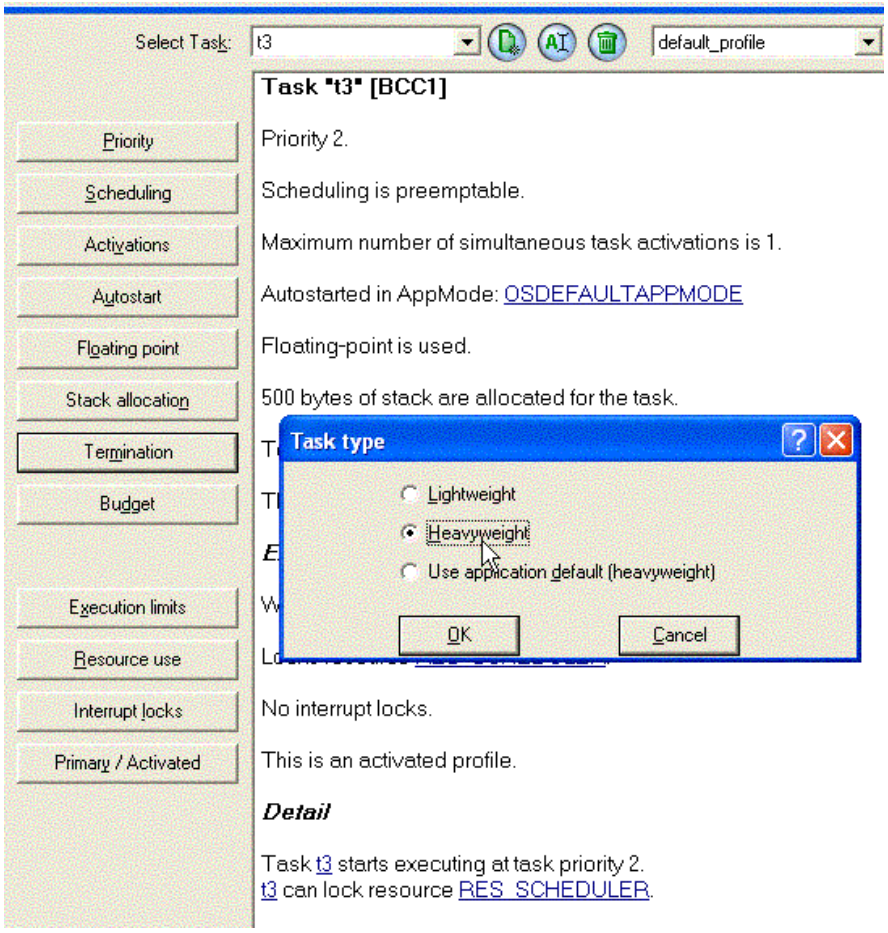


Figure 4:6 - Setting Default Termination Type

Important: To take advantage of the lightweight task optimization, you must include the correct task header file (called `<TaskID>.h`) when compiling the task. The generic `osek.h` or `oseklib.h` file will not give you the expected behavior. It is important that you do not configure a task to use lightweight termination and then include a generic header file.

4.6 Simple Task Activation

A task can only run if it is **activated**. Activation either moves a task from the *suspended* state into the *ready* state or it adds another entry to the queue of *ready* tasks. If a task becomes the highest priority *ready* task, it moves into the *running* state and RTA-OSEK Component calls the task's entry function.

An activation count must be specified for all tasks. When multiple activations are specified, RTA-OSEK Component treats the task as being BCC2.

If a task is activated, but it is not in the *suspended* state, RTA-OSEK Component will queue the task activations and the task will run once for each of the activations. It is an error to exceed the activation count and your application will generate `E_OS_LIMIT` errors when this happens (even in the Standard build).

Tasks can be activated from both tasks and ISRs. When you activate a task from an ISR, the task will only start *running* once the ISR has completed and once any higher priority tasks that were *ready* or *running* have terminated.

When you activate a task from another task, the behavior depends upon the relative task priorities. In general, if the activated task has higher priority than the task doing the activation, then the newly activated task will preempt the current task*. Otherwise, it will wait until the current task terminates and it becomes the highest priority *ready* task.

Figure 4:7 shows how this preemption works. In this example, `Task1` is *running*, but it is preempted by a higher priority task called `Task2`. `Task2` executes to completion and then `Task1` resumes from the point that it was preempted, until it finishes.

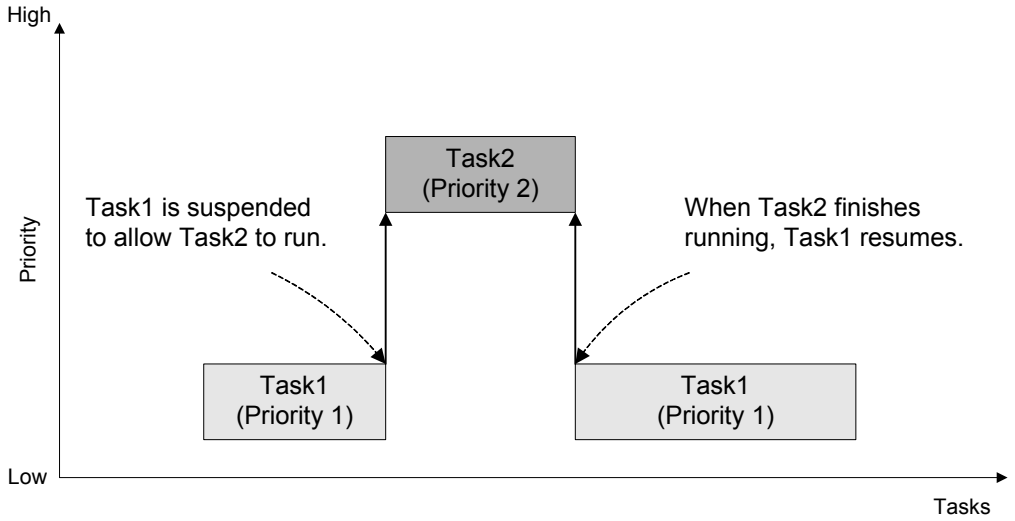


Figure 4:7 - Preemption of a Running Task by a Higher Priority Task

In a well-designed real-time system, it is unusual for a task to activate a higher priority task. Normally it is the ISRs that react to the incoming stimuli. They then activate the tasks that implement the responses. In turn, the tasks may

* In fact, if the calling task has a resource locked, the activated task has to be of higher priority than the highest priority task locking the resource. Also, if the calling task is non-preemptive, the activated task will not preempt.

activate lower priority tasks to implement the responses that have longer deadlines.

Observing this fact leads to one of the major optimizations in RTA-OSEK Component. If you specify that your tasks never activate a higher priority task, RTA-OSEK Component can eliminate a large amount of internal code. It can do this because it never has to test within an API call to see if preemption should occur.

Tasks can be activated in a number of different ways. The advanced techniques are listed in Section 4.12, but the basic mechanism for task activation is the `ActivateTask(TaskID)` API call, which directly activates a task.

The `ActivateTask()` call places the named task into the *ready* state. It is important to note that activating a task does not cause the task to execute. It simply moves the task into the *ready* state. RTA-OSEK Component is responsible for placing the task into the *running* state.

4.7 The Idle Task

Any preemptive operating system must have something to do when there are no tasks or ISRs to run. In OSEK this is achieved by an **idle mechanism**. The idle mechanism is implemented in RTA-OSEK Component using an **idle task**, called `osek_idle_task`.

The idle task has many of the features of a normal task, but it cannot be activated, terminated or chained.

It has the lowest priority of any task in the system. It cannot use any internal resources because it would never release them to allow other tasks to run (remember that the idle task never terminates).

The idle task can use standard, linked or message resources and it does not count towards the maximum number of user tasks (usually 16 or 32) available on your target hardware.

RTA-OSEK generates a special task-specific header file called `osekmain.h` for the idle task. This should normally be included in the file containing the idle task code.

The code that implements the idle task is the code that executes after `StartOS()` returns. Normally, this is the code in the application startup function. Code Example 4:5 shows an example startup function.

```
#include "osekmain.h"
OS_MAIN(main)
{
  /* System hardware initialization. */
  StartOS(OSDEFAULTAPPMODE);
  for (;;) {
    /* This loop body is the osek_idle_task. */
  }
}
```

Code Example 4:5 - An Application Startup Function

The idle task can wait for events. If you use the idle task to wait for events, rather than any other extended task, the OS overhead is much lower.

4.8 Multiple Activation

Under most circumstances you will only activate a task when it is in the *suspended* state. However, you may need to implement a system where the same task must be activated a number of times and where the shortest time between successive activations is less than the time needed to run the task.

If this happens you will be activating the task while it is in the *ready* state or the *running* state. This means that activations will be lost.

To prevent loss of activations, you must specify the maximum number of multiple activations required for the task.

Important: In accordance with the OSEK OS standard, this feature is only available for basic tasks. Remember that you cannot specify multiple activations for extended tasks.

You will use the RTA-OSEK GUI to specify the maximum number of simultaneous task activations. Figure 4:8 shows that, for the task in this example, the maximum number of activations has been set to 10.

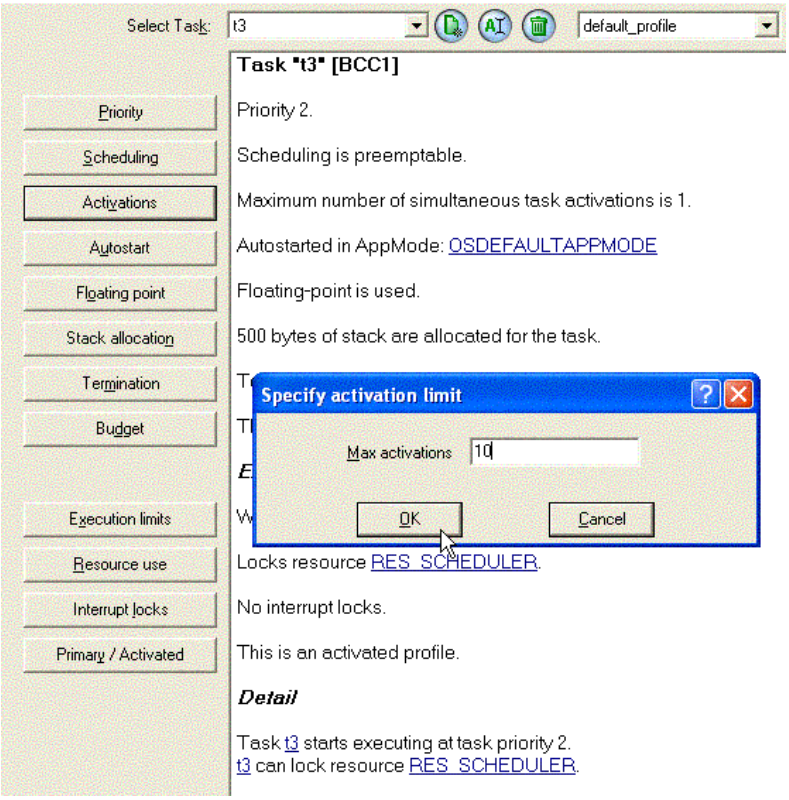


Figure 4:8 - Specifying the Maximum Number of Activations

When you perform analysis on your application, RTA-OSEK will calculate the maximum size of the multiple activation queue needed for each BCC2 task.

4.9 Non-Preemptive Tasks

In general, a task can be preempted by a task of higher priority. You can prevent other tasks from preempting it by declaring the task to be non-preemptable in the RTA-OSEK GUI[†].

If tasks are declared as non-preemptive, once they move to the *running* state they run to completion and then terminate (unless they make a `Schedule()` call, explained in Section 4.9.1). Non-preemptive tasks can still be interrupted by ISRs.

You will often find that it is unnecessary to use non-preemptable tasks because there are other, more suitable methods, which you can use to achieve the same effect. If you use these other techniques, it will usually result in a more responsive system. You will find out more about these techniques later, but they include:

- Using standard resources to serialize access to data or devices.
- Using internal resources to specify exactly which other tasks cannot cause preemption.

4.9.1 Rescheduling

When a task is running non-preemptively it prevents any task (including those of higher priority) from executing. Sometimes, however, it is useful for non-preemptive tasks to offer explicit places where rescheduling can take place. This is more efficient because higher priority tasks can have shorter response times to system stimuli.

A system where all tasks run non-preemptively and offer points for rescheduling is known as a **co-operatively scheduled** system.

The `Schedule()` API call can be used to momentarily remove the preemption constraints imposed by both the non-preemptive tasks and the tasks using internal resources.

When `Schedule()` is called, any *ready* tasks that have a higher priority than the calling task are allowed to run. `Schedule()` does not return until all higher priority tasks have terminated.

This feature is of no use in a fully preemptive system. If you do not intend to use it, you can disallow calls to `Schedule()` in the RTA-OSEK GUI using the Application Optimizations. Figure 4:9 shows how the `Schedule()` call can be disallowed.

[†] The non-preemptive task itself, however, can call the `Schedule()` API call, which will cause a task switch if a higher priority task is ready to execute.

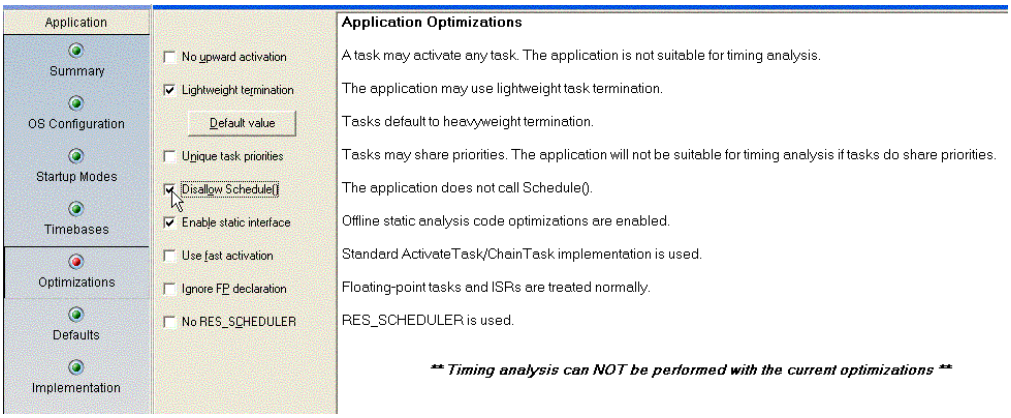

Application
Summary
OS Configuration
Startup Modes
Timebases
Optimizations
Defaults
Implementation
No upward activation
Lightweight termination
Default value
Unique task priorities
Disallow Schedule()
Enable static interface
Use fast activation
Ignore FP declaration
No RES_SCHEDULER
Application Optimizations
A task may activate any task. The application is not suitable for timing analysis.
The application may use lightweight task termination.
Tasks default to heavyweight termination.
Tasks may share priorities. The application will not be suitable for timing analysis if tasks do share priorities.
The application does not call Schedule().
Offline static analysis code optimizations are enabled.
Standard ActivateTask/ChainTask implementation is used.
Floating-point tasks and ISRs are treated normally.
RES_SCHEDULER is used.
** Timing analysis can NOT be performed with the current optimizations **

Figure 4:9 - Disallowing Calls to Schedule()

If you disallow calls to `Schedule()` in the RTA-OSEK GUI, you will see the following benefits:

- The worst-case stack requirement is reduced if `Schedule()` is not called.
- Timing analysis is available. (You cannot currently perform timing analysis on systems that call `Schedule()`.)

4.10 Fast Activation

RTA-OSEK Component checks that the activation limit for the task is not exceeded each time that you make an API call which results in task activation. If this limit is exceeded, the `E_OS_LIMIT` error is raised. This check, however, consumes a lot of time.

If you use the RTA-OSEK Planner to verify that your application is schedulable you are, in fact, showing offline that `E_OS_LIMIT` will never be raised at run-time. Because of this, RTA-OSEK allows you to use fast task activation that doesn't need to make the `E_OS_LIMIT` check.

Fast task activation is selected in the RTA-OSEK GUI using the Application Optimizations. In Figure 4:10, the system has been set to use fast task activation.

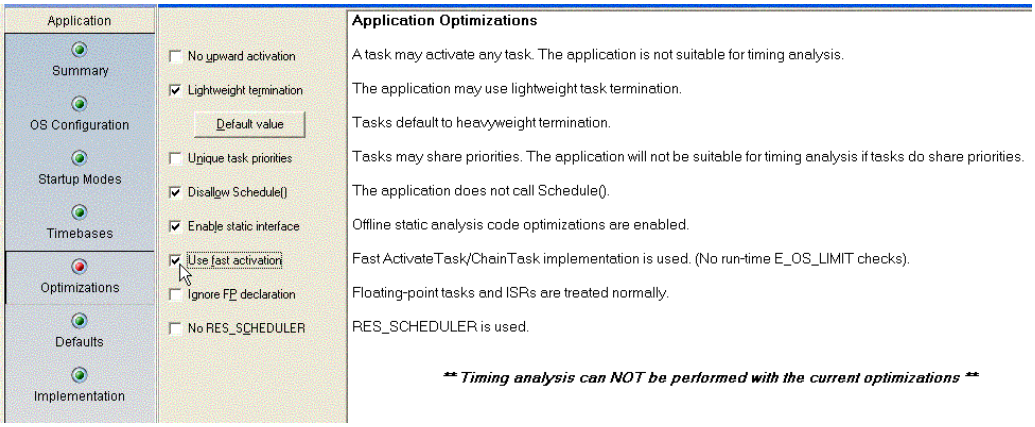

Application
Summary
OS Configuration
Startup Modes
Timebases
Optimizations
Defaults
Implementation
No upward activation
Lightweight termination
Default value
Unique task priorities
Disallow Schedule()
Enable static interface
Use fast activation
Ignore FP declaration
No RES_SCHEDULER
Application Optimizations
A task may activate any task. The application is not suitable for timing analysis.
The application may use lightweight task termination.
Tasks default to heavyweight termination.
Tasks may share priorities. The application will not be suitable for timing analysis if tasks do share priorities.
The application does not call Schedule().
Offline static analysis code optimizations are enabled.
Fast ActivateTask/ChainTask implementation is used. (No run-time E_OS_LIMIT checks).
Floating-point tasks and ISRs are treated normally.
RES_SCHEDULER is used.
** Timing analysis can NOT be performed with the current optimizations **

Figure 4:10 - Using Fast Task Activation

4.11 Floating-Point

Floating-point calculations are relatively time consuming to perform purely in software. They are also expensive, in terms of silicon, to implement in hardware.

As a result of this, very few embedded systems make full use of floating-point calculations. RTA-OSEK, therefore, generally assumes that floating-point is not used in an application.

If any of your tasks or ISRs use floating-point hardware, you must make sure that this is declared in the RTA-OSEK GUI. RTA-OSEK Component will then make sure that any hardware registers that are involved with floating-point calculations are saved and restored on entry/exit for these tasks and ISRs.

RTA-OSEK is able to calculate exactly how much memory to reserve in order to save your floating-point tasks and ISRs. It knows this because it can work out the worst-case preemption depth for tasks and ISRs that use floating-point.

You can see the results of the calculation in the stack depth analysis in the RTA-OSEK GUI.

Figure 4:11 shows a stack analysis example where `t1`, `t2` and `t3` use floating-point.

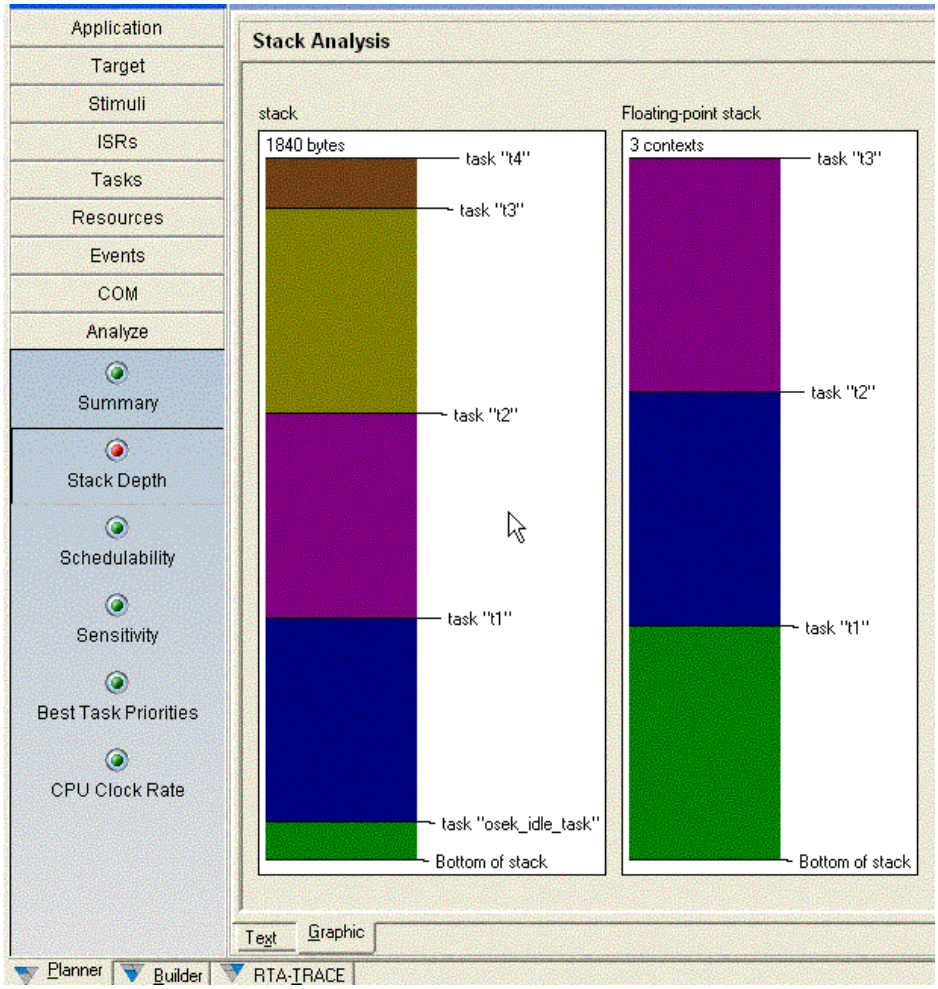


Figure 4:11 - Stack Depth Analysis Example

Two floating-point save context areas are set aside because RTA-OSEK Component does not need to perform a save in the lowest priority task.

If you only have one task or ISR that uses floating-point, for example, then no floating-point saves or restores are needed and RTA-OSEK Component imposes no floating-point overhead on the application at all.

4.11.1 Customizing Floating-Point Operation

Each target processor supported by RTA-OSEK is provided with support for floating-point tasks and ISRs. Some targets have software support for floating-point implementations and others have support for hardware implementations.

Some target hardware may not have the same floating-point implementation that is supported by RTA-OSEK. For example, your target may have an off-chip floating-point coprocessor. To overcome this, RTA-OSEK is supplied with two source files that you can modify and link with your application.

The source files can be used to change how the floating-point save and restore is performed. The files are called `osfptgt.c` and `osfptgt.h`. You will find them in the `<install dir>\<target>\inc` folder.

4.12 Advanced Task Activation

In Section 4.6 you learnt how the `ActivateTask()` call could be used to activate a task. You will now see the other methods that are available.

4.12.1 Direct Activation

Direct activation is where an API call is made to explicitly move a task from a *suspended* to a *ready* state or to queue task activations where multiple task activations are permitted.

API Call Name	Description	Static Version
`ActivateTask()`	A task or ISR can make this call to activate the task directly.	`ActivateTask_TaskID()` The static version allows RTA-OSEK to optimize the code generated, based on the relative priority of the caller and activated task.
`ActivateTaskset()`	A task or ISR can make this call to activate each task in the taskset.	`ActivateTaskset_TasksetID()` The static version allows RTA-OSEK to optimize the code generated, based on the relative priority of the caller and activated tasks.

API Call Name	Description	Static Version
`ChainTask()`	A task can make this call to terminate the currently running task and to activate the task indicated.	`ChainTask_TaskID()`
`ChainTaskset()`	A task can make this call to terminate the currently running task and to activate the tasks in the taskset indicated.	`ChainTaskset_TasksetID()`

4.12.2 Activation by Event

For each event in the system, you can specify task(s) that are activated each time the event occurs.

4.12.3 Activation by a Message

For each message in the system, you can specify a task that is activated each time the message is sent.

4.12.4 Activation by an Alarm

For each alarm in the system, you can specify a task that is activated each time the alarm expires.

4.12.5 Activation by a Periodic Schedule

When you create a periodic schedule, a periodic activation pattern is specified for one or more tasks. RTA-OSEK Component ensures that each task is activated according to the pattern specified.

4.12.6 Activation by a Planned Schedule

When you create a planned schedule, you will specify a specific activation pattern for one or more tasks. RTA-OSEK Component ensures that each task is activated according to the pattern specified.

4.12.7 Auto-Activation

Tasks can be **auto-activated**, which means that when the operating system starts, they are activated automatically during `StartOS()`. Auto-activation is often used for tasks that wait on events because it removes the need to write code to activate the tasks. The RTA-OSEK GUI can be used to specify that a task is only auto-activated in specific application modes – choose the application mode in question and select the tasks you want to autoactive in it.

In Figure 4:12, `t2` and `t3` are autostarted in the default application mode.

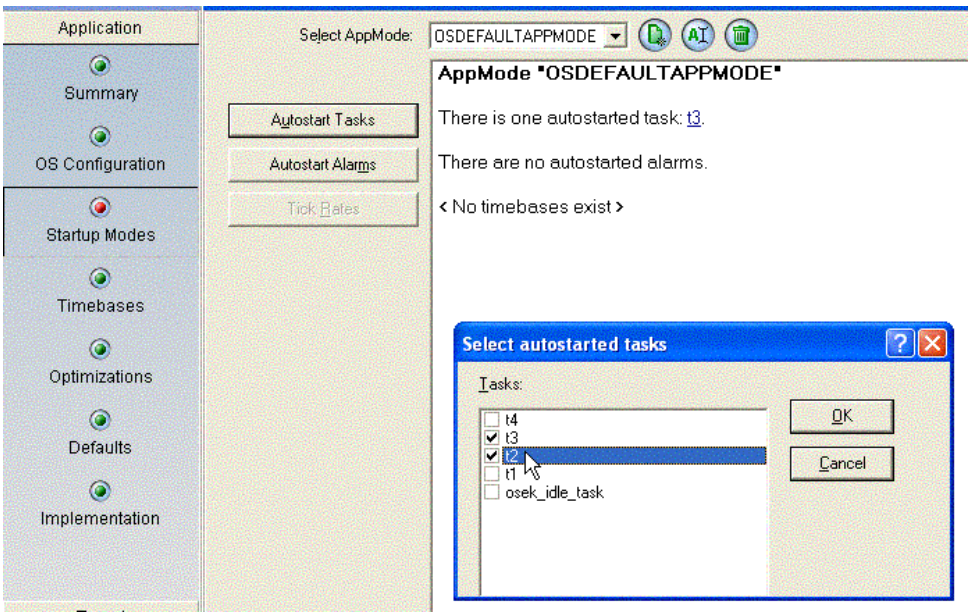


Figure 4:12 - Auto-Activating Tasks in Application Modes

4.13 Tasksets

RTA-OSEK allows you to use **tasksets**. They provide an efficient way of activating several tasks at the same time.

Portability: Tasksets are an enhancement available in RTA-OSEK, but are not part of the OSEK standard.

A taskset is simply a named collection of tasks. Any task, except the idle task, can belong to a taskset. Each task can also belong to more than one taskset.

Tasksets are declared in the RTA-OSEK GUI. Figure 4:13 shows you that `ts1` has been declared and that `t2` and `t4` belong to it.

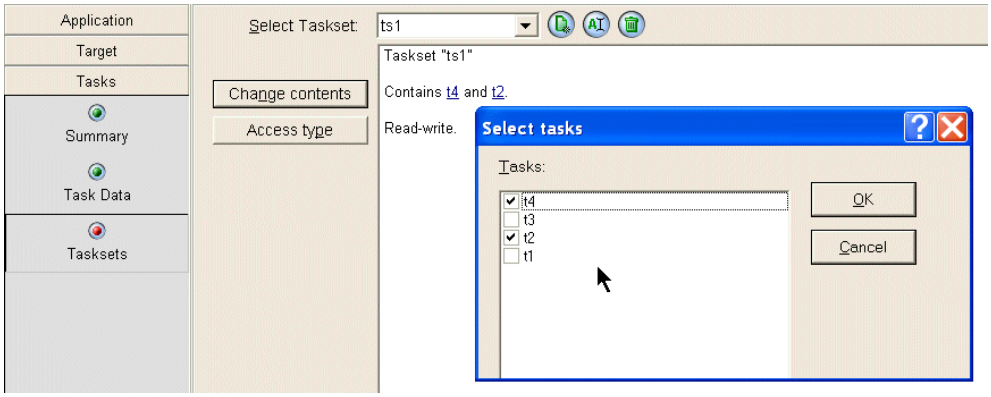


Figure 4:13 - Configuring a Taskset

You can set the taskset access type to **read-only** or **read-write**. The contents of the taskset can be modified at run-time if you set the access type to read-write.

If you want to find out more about the RTA-OSEK Component API calls that are used to manipulate tasksets, have a look at the *RTA-OSEK Reference Guide*.

4.13.1 Predefined Tasksets

RTA-OSEK always generates the following predefined read-only tasksets.

Taskset	Description
`os_all_tasks`	Contains all the tasks in the system.
`osek_cc2_tasks`	Contains all the BCC2 and ECC2 tasks.
`osek_ecc_tasks`	Contains all the ECC1 and ECC2 tasks.
`os_no_tasks`	Contains no members.
`os_ready_tasks`	Contains all tasks in the *ready* state and the *running* task.

These tasksets are useful when manipulating read-write tasksets. They allow you to do things like membership tests to clear tasksets.

4.14 Choice of Task Characteristics

RTA-OSEK is designed to be very aggressive at minimizing code and data usage on the target application. It will analyze the characteristics of the application and generate a system containing only the features that are required.

Your choice of task characteristics has a major influence on the final application size and speed. Tasksets with BCC2 tasks, for instance, are very inefficient.

If you want to create the most efficient application, your system should contain BCC1 tasks exclusively, it should use lightweight termination and should not use floating-point.

As you add features to your application, the system will inevitably become slightly larger and slower.

A system with one or more BCC2 tasks has a greater overhead than one with only BCC1 tasks. A system without shared priorities, even if multiple activations are allowed, will be more efficient than one with shared priorities.

A system with ECC1 tasks has an even greater overhead still and a system with one or more ECC2 tasks has the largest overhead of all.

4.15 Controlling Task Execution Ordering

In many cases you will need to constrain the execution order of specific tasks. This is particularly true in data flow based designs where one tasks needs to perform some calculation before another task uses the calculated value.

If the execution order is not constrained, a **race condition** may occur and the application behavior will be unpredictable. Task execution ordering can be controlled in the following ways:

- Direct activation chains (see Section 4.15.1).
- Priority levels (see Section 4.15.2).
- Non-preemptable tasks (see Section 4.9).

4.15.1 Direct Activation Chains

When you use direct activation chains to control the execution order, tasks make `ActivateTask()` calls on the task(s) that must execute following the task making the call.

Let's look at an example. There are three tasks `Task1`, `Task2` and `Task3` that must execute in the order `Task1`, then `Task2`, then `Task3`.

In this example, you would write task bodies like the ones in Code Example 4:6.

```
#include "Task1.h"
TASK(Task1) {

  /* Task1 functionality. */
  ActivateTask(Task2);
  TerminateTask();
}
```

```
#include "Task2.h"
TASK(Task2) {

  /* Task2 functionality. */
  ActivateTask(Task3);
  TerminateTask();
}
```

```
#include "Task3.h"
TASK(Task3) {

  /* Task3 functionality. */
  TerminateTask();
}
```

Code Example 4:6 - Using Direct Activation Chains

To make the application suitable for timing analysis, you must be certain that all task activations are downward. In other words, you must ensure that tasks only activate lower priority tasks.

4.15.2 Using Priority Levels

The priority level approach to constraining task execution ordering can be used to exploit the nature of the preemptive scheduling policy to control activation order.

Remember that you learnt, in Section 4.1, that under fixed priority preemptive scheduling the scheduler would always run the highest priority task. If a number of tasks are released onto the ready queue, they will execute in priority order. This means that you can use task priorities to control execution order.

Following on from our previous example, in Code Example 4:6, let's assume that `Task1` has the highest priority and `Task3` has the lowest priority. This means that the task bodies can be rewritten to exploit priority level controlled activation.

This can be seen in Code Example 4:7.

```
#include "Task1.h"
TASK(Task1) {
  /* Task1 functionality. */
  ActivateTask(Task2);  /* Runs when Task1
                         * terminates. */
  ActivateTask(Task3);  /* Runs when Task2
                         * terminates. */
  TerminateTask();

}
```

```
#include "Task2.h"
TASK(Task2) {
  /* Task2 functionality. */
  TerminateTask();

}
```

```
#include "Task3.h"
TASK(Task3) {
  /* Task3 functionality. */
  TerminateTask();

}
```

Code Example 4:7 - Using Priority Level Controlled Activation

This method of control is even more useful for tasksets, where you could make a single call to simultaneously activate `Task1`, `Task2` and `Task3`.

If you use automatic priority allocation, the priorities that you have specified can be changed. This can affect priority level controlled activation.

To make sure that the relative ordering of priorities is maintained, you can specify that a task has a set of required lower priority tasks. These are the tasks that must execute after the selected task.

Figure 4:14 shows that for `t4`, the required lower priority tasks are `t3` and `t1`.

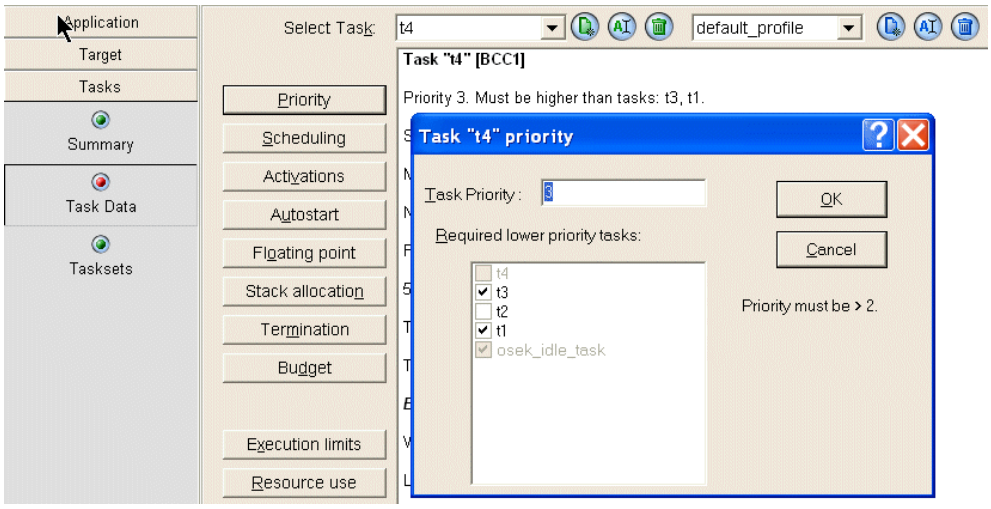


Figure 4:14 - Setting the Required Lower Priority Tasks

4.16 Synchronization with Basic Tasks

Basic tasks can only synchronize at the start or end of task execution. If other synchronization points are required, you must implement them yourself.

For example, if a task is built as a state machine (using a C switch statement, for instance) then you can set a state variable, issue a `TerminateTask()`

call and wait for re-activation. Code Example 4:8 shows how this can be achieved.

```
#include "Task1.h"

int State;

TASK(Task1) {

  switch (State) {

    case 0:
      /* Synchronization point 0. */
      State = 1;
      break;

    case 1:
      /* Synchronization point 1. */
      State = 2;
      break;

    case 2:
      /* Synchronization point 2. */
      State = 0;
      break;
  }

  TerminateTask();

}
```

Code Example 4:8 - Multiple Synchronization Points in a Basic Task

4.17 Simulating Waiting using Basic Tasks

You may need to build a system containing only basic tasks where those tasks need to wait on some event (remember that in Section 4.2.2 you learnt that only extended tasks can wait on events).

You can simulate this type of functionality in your application using tasksets. When you do this the taskset becomes a pseudo-event.

You can see an example of this in Code Example 4:9.

```
#include "Task1.h"
TASK(Task1) {

  TasksetType Tmp;
  /* Create a singleton set holding this task. */
  GetTasksetRef(Task1, &Tmp);
  /* Subscribe. */
  MergeTaskset(DataReady, Tmp);
  TerminateTask();
}
```

```
#include "Task2.h"
TASK(Task2) {
  /* Process Data. */
  /* Notify any waiting tasks. */
  ActivateTaskset(DataReady);
  AssignTaskset(DataReady, os_no_tasks);
  TerminateTask();

}
```

Code Example 4:9 - Subscribing to a Pseudo-Event

In this example, `Task1` needs to be informed when `Task2` has completed data processing. `Task1` subscribes to a `DataReady` taskset. When `Task2` has processed the data it notifies all tasks waiting on the data by activating the `DataReady` taskset.

Tasks Summary

- A task is a concurrent activity.
- There are two classes of tasks: basic and extended. Each class has two levels: level 1 and level 2.
- Tasks are scheduled according to priority. When a higher priority task is made *ready* to run it will preempt lower priority tasks.
- Tasks exist in states: *ready, running*, *suspended* or *waiting (however, the waiting state exists for extended tasks only).*
- If a task terminates, it must call `TerminateTask()`, `ChainTask(TaskID)` or `ChainTaskset(TasksetID)` to do so.
- You must include the correct task specific header (`<TaskID>.h`) with your application code. For this reason, it is best to put each task in a separate source file.
- Tasks can only be activated when they are in the *suspended* state unless you specify multiple activations.
- Tasksets are a special feature provided in RTA-OSEK to allow you to activate several tasks simultaneously. In this case, `ChainTaskset(TaskID)` becomes an alternative way of terminating a task.

5 Interrupts

Interrupts provide the interface between your application and the things that happen in the real-world. You could, for example, use an interrupt to capture a button being pressed, to mark the passing of time or to some capture some other stimulus.

When an interrupt occurs, the processor usually looks at a predefined location in memory called a **vector**. A vector usually contains the address of the associated interrupt handler. Your processor documentation and the *RTA-OSEK Binding Manual* for your target will give you further information on this. The block of memory that contains all the vectors in your application is known as the **vector table**.

5.1 Single-Level and Multi-Level Platforms

Target processors are categorized according to the number of interrupt priority levels that are supported*. You should make sure that you fully understand the interrupt mechanism on your target hardware.

There are two different types of target:

- Single-level.
  On single-level platforms there is a single interrupt priority. If an interrupt is being handled, all other pending interrupts must wait until current processing has finished.
- Multi-level.
  On multi-level platforms there are multiple interrupt levels. If an interrupt is being handled, it can be pre-empted by any interrupt of higher priority.

5.2 Interrupt Service Routines

OSEK operating systems capture interrupts using **Interrupt Service Routines** (ISRs). ISRs are similar to tasks; however, ISRs differ because:

- They cannot be activated by RTA-OSEK Component API calls.
- The terminating APIs are not called.
- They cannot appear in tasksets.
- They start executing from their entry point at the associated interrupt priority level.
- Only a subset of the RTA-OSEK Component API calls can be made. (To call an RTA-OSEK Component API call from within an ISR, refer to the function's calling environment in the *RTA-OSEK Reference Guide*.)

* Make sure that you don't confuse interrupt priority levels on the target processor with the priority of tasks.

5.2.1 Category 1 and Category 2 Interrupts

OSEK operating systems classify interrupts into two categories called **Category 1** and **Category 2**. The category indicates whether or not the OS is involved with handling the interrupt.

Category 1 Interrupts

Category 1 interrupts do not interact with RTA-OSEK Component. They should always be the highest priority interrupts in your application. It is up to you to configure the hardware correctly, to write the handler and to return from the interrupt.

You can find out about Category 1 interrupt handlers in Section 5.5.1.

The handler executes at or above the priority level of RTA-OSEK Component. However, you can make RTA-OSEK Component API calls for enabling/disabling and resuming/suspending interrupts.

Category 2 Interrupts

With Category 2 interrupts, the interrupt vector points to internal RTA-OSEK Component code. When the interrupt is raised, RTA-OSEK Component executes the internal code and then calls the handler that you have supplied.

The handler is provided as an ISR bound to the interrupt (which you can think of as a very high priority task). Execution starts at the specified entry point of the ISR and continues until the entry function returns. When the entry function returns, RTA-OSEK Component executes another small section of internal code and then returns from the interrupt.

Figure 5:1 shows the state diagram for a Category 2 interrupt handler.

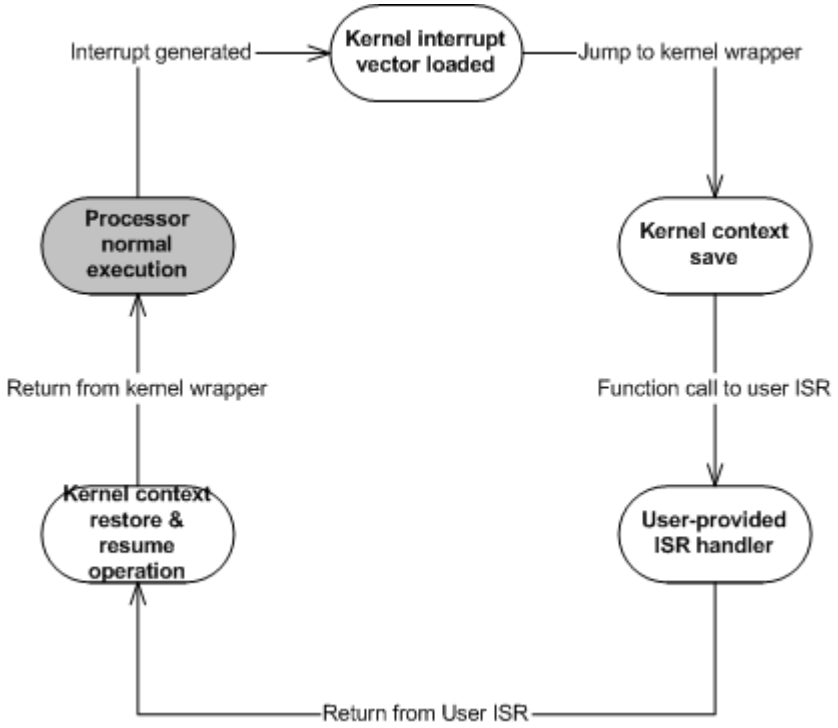


Figure 5:1 - Category 2 Interrupt Handling State Diagram

Figure 5:2 shows how the internal RTA-OSEK Component code wrappers can be visualized.

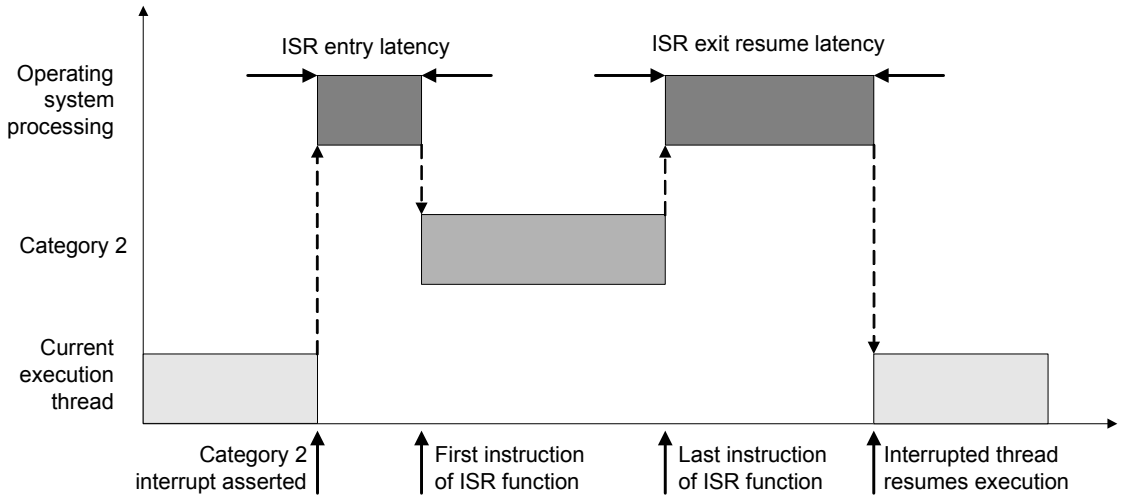


Figure 5:2 - Visualizing RTA-OSEK Component Category 2 Wrappers

5.3 Interrupt Priorities

Interrupts execute at an **interrupt priority level** (IPL). It is important that you don't confuse IPLs with task priorities. An IPL of 1 is higher than the highest task priority used in your application.

The IPL is a processor-independent description of the interrupt priority on your target hardware. The *RTA-OSEK Binding Manual* for your target will tell you more about how IPLs are mapped onto target hardware interrupt priorities.

ISRs can be nested (assuming that the processor supports interrupt nesting). So, for example, a higher priority ISR can interrupt the execution of a low priority ISR. However, an ISR can never be preempted by a task.

A Category 1 interrupt handler must never be interrupted by a Category 2 interrupt. In other words, you must not have a Category 2 interrupt with a higher interrupt priority than a Category 1 interrupt.

The interrupt priority hierarchy is illustrated in Figure 5:3.

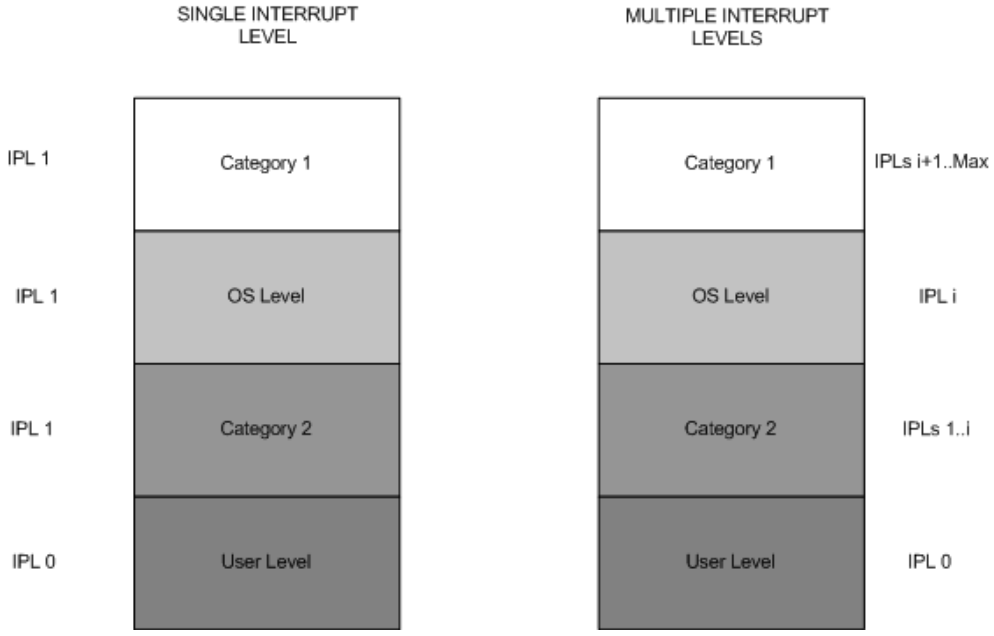


Figure 5:3 - The Interrupt Priority Hierarchy

5.3.1 OS Level

The highest priority Category 2 interrupt defines **OS level**. If execution occurs at OS level, or higher, then no other Category 2 interrupt can occur.

RTA-OSEK Component uses OS level to guard against concurrent access to internal OS data structures. If a task executes at OS level then no RTA-OSEK Component operations will take place (except for calls made by the task).

5.3.2 User Level

User level is the lowest interrupt priority level that allows all interrupts to be handled. All tasks start executing at user level from their entry point.

A task will sometimes need to run above user level, so that it can access data shared with an interrupt handler. While the data is being accessed it must prevent the interrupt being serviced.

An ISR may preempt a task even when the task is running with interrupt priority level above user level. It can only do this, however, if the ISR has a higher interrupt priority level than the current level.

5.4 Simple Interrupt Configuration

In RTA-OSEK Component, interrupts are configured statically using the RTA-OSEK GUI. Figure 5:4 shows how an interrupt has been constructed.

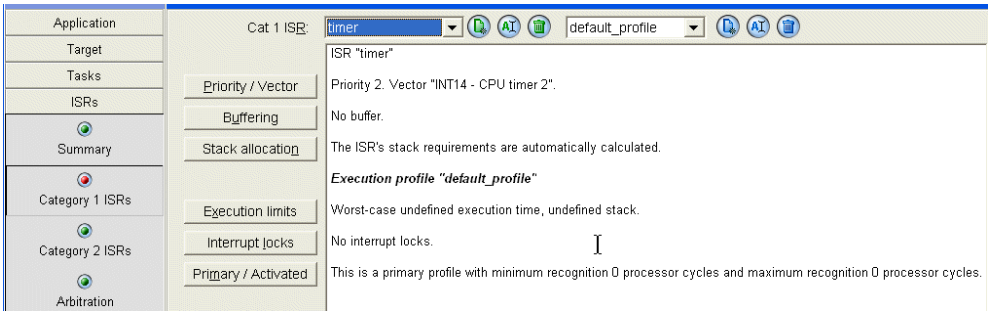


Figure 5:4 - Configuring an Interrupt using the RTA-OSEK GUI

At the simplest level, an interrupt has the following attributes:

- An interrupt name.
  The name is used to refer to C code that you will write to implement the handler functionality (you will learn how to do this in Section 5.5).
- An interrupt priority.
  The priority is used by the scheduler to determine when the interrupt runs (in a similar way to a task priority being used for tasks). Note that some targets only support a single interrupt priority.

Important: You must make sure that the programmed priority level of an interrupting device agrees with the level configured in the RTA-OSEK GUI.

- An interrupt vector.
  RTA-OSEK uses the specified vector to generate the vector table entry for the interrupt.

The meaning of vector and priority is processor-specific. You will need to check the *RTA-OSEK Binding Manual* for your specific target.

In most cases, RTA-OSEK can generate the vector table automatically. It will create a vector table with the correct vectors pointing to the internal wrapper code. The vector table generated by RTA-OSEK is output to the `osgen` file.

If you want to write your own vector table then you must make sure that RTA-OSEK does not generate a vector table itself. You can prevent a vector table being generated using the Target Vectors settings, shown in Figure 5:5.

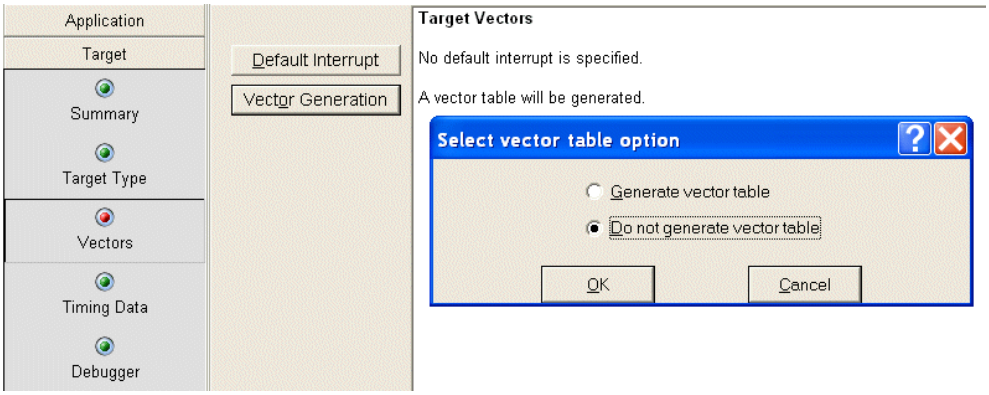


Figure 5:5 - Preventing RTA-OSEK from Automatically Generating a Vector Table

The *RTA-OSEK Binding Manual* for your target will explain how to provide your own vector table.

5.5 Providing Interrupt Handlers

You will now learn about interrupt handlers for Category 1 and Category 2 interrupts.

5.5.1 Category 1 Interrupt Handlers

Generally, the binding of Category 1 interrupt handlers is non-portable. You will usually write these using compiler-specific extensions to ANSI C. Some compilers, however, cannot do this. When this happens you will need to write an assembly language handler.

You must make sure that the name of a Category 1 ISR entry function is the same as the name that you specified for the ISR during configuration.

For Category 1 ISRs, there is usually a compiler-specific keyword that has to be used when defining entry functions.

An entry function for a Category 1 ISR is shown in Code Example 5:1.

```
interrupt void Interrupt1(void) {
  /* Handler body. */
  /* Return from interrupt. */
}
```

Code Example 5:1 - Entry Function for a Category 1 ISR

You will find any target specific information in the *RTA-OSEK Binding Manual* for your target.

5.5.2 Category 2 Interrupt Handlers

You saw earlier that Category 2 interrupts are handled under the control of RTA-OSEK Component. A Category 2 interrupt handler is similar to a task. It has an entry function that is called by RTA-OSEK Component when the interrupt handler needs to run. A Category 2 interrupt handler is written using the C syntax in Code Example 5:2.

```
ISR (isr identifier){ … }    /* Category 2 ISR
                              * entry function. */
```

Code Example 5:2 - Category 2 Interrupt Handler

Code Example 5:3 shows the code for a simple interrupt handler called `Interrupt1`.

```
#include "Interrupt1.h"  /* Header file generated
                          * by RTA-OSEK. */

ISR (Interrupt1) {

  DismissInterrupt();  /* User supplied function to
                        * cancel interrupt. */
  ActivateTask_Task1();/* Let Task1 do the work. */

}
```

Code Example 5:3 - Entry Function for a Category 2 ISR

Important: You do not need to provide any C function prototypes for Category 2 ISR entry functions. These are provided in the header file that is generated by RTA-OSEK. The appropriate file for each ISR should be included because it contains declarations that are specific to the named handler. ISRs should be in separate source files for this reason.

5.6 Enabling and Disabling Interrupts

Interrupts will only occur if they are enabled. By default, RTA-OSEK Component ensures that all interrupts are enabled when `StartOS()` returns.

You will often need to disable interrupts for a short amount of time to prevent interrupts occurring in a **critical section** of code in either tasks or ISRs. A critical section is a sequence of statements that accesses shared data.

You can enable and disable interrupts using a number of different API calls:

- `DisableAllInterrupts()` and `EnableAllInterrupts()`
  Disable and enable all interrupts that can be disabled on the hardware (usually all those interrupts that can be masked).
  These calls cannot be nested.
- `SuspendAllInterrupts()` and `ResumeAllInterrupts()`
  Suspend and resume all interrupts that can be disabled on the hardware (usually all those interrupts that can be masked).
  These calls can be nested.
- `SuspendOSInterrupts()` and `ResumeOSInterrupts()`
  Suspend and resume all Category 2 interrupts on the hardware.
  These calls can be nested.

Important: You must make sure that there are never more '`Resume`' calls than '`Suspend`' calls. If there are, it can cause serious errors and the behavior is undefined. Subsequent '`Suspend`' calls may not work. This will result in unprotected critical sections.

Code Example 5:4 shows you how the interrupt control API calls are used and nested correctly.

```
#include "Task1.h"
TASK(Task1) {

  DisableAllInterrupts();
  /* First critical section, nesting not allowed.*/

  EnableAllInterrupts();
  SuspendOSInterrupts();

  /* Second critical section, nesting allowed. */

  SuspendAllInterrupts();

  /* Third critical section. */

  ResumeAllInterrupts();
  ResumeOSInterrupts();
  TerminateTask();
}
```

Code Example 5:4 - Nesting Interrupt Control API Calls

In the case of Category 1 interrupts, you must make sure that no RTA-OSEK Component API calls are made (except for other `Suspend`/`Resume` calls) for the entire time that the interrupts are disabled.

If a Category 2 ISR raises the interrupt level above OS level, it may not make any other RTA-OSEK Component API calls, except for the appropriate call to restore the interrupt priority. When executing an ISR, you are not allowed to lower the interrupt priority level below the initial level.

Advanced Interrupt Concepts

5.7 Getting Resources

The `Enable`/`Disable` and `Suspend`/`Resume` calls, which you learnt about earlier, are interrupt manipulation facilities that provide a coarse control mechanism. Using these calls you can either prevent Category 2 interrupts or prevent all maskable interrupts.

In some cases you may want to improve the granularity of control, especially if you are using target hardware that supports multiple interrupt levels. You might, for example, share data with some interrupts and only want to prevent those interrupts occurring.

RTA-OSEK Component supports OSEK **combined resources** that can be shared by both tasks and ISRs.

A combined resource can be held using the `GetResource(ResourceID)` API call. When this happens, interrupts are suspended if their priority is less than or equal to the highest priority interrupt sharing the resource. When the resource is released, using `ReleaseResource(ResourceID)`, the suspended interrupts are resumed. You can use this feature to suspend subsets of ISRs.

5.8 Interrupt Arbitration

If multiple interrupts share the same interrupt priority level, you must define an **arbitration order** in the RTA-OSEK GUI. This is used for analysis.

The arbitration order is used to specify the order in which interrupts of the same priority are serviced when two or more are pending simultaneously.

Figure 5:6 shows how the interrupt arbitration order is defined.

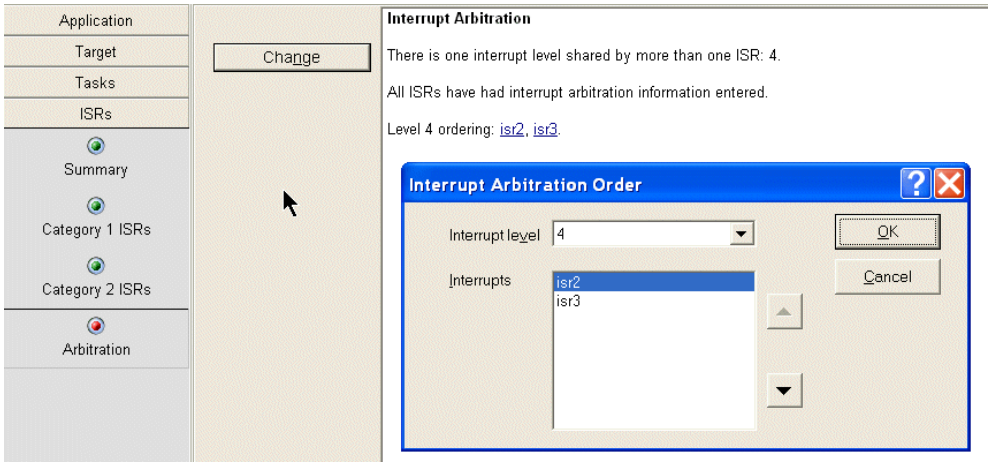


Figure 5:6 - Defining the Arbitration Order

In the example in Figure 5:6, you will see that the arbitration order specifies that `ISR_1` be serviced first if both interrupts are pending simultaneously.

For many processors, the interrupt arbitration order is fixed (details can be found in the processor reference manual). For other processors, you can define the arbitration order.

Important: You should make sure that the arbitration order specified matches the arbitration order of the interrupts within your application. Ensure that the information given in the configuration file correctly describes the run-time behavior of the interrupts.

5.9 Using Floating-Point

You saw earlier that floating-point calculations are relatively time consuming to perform purely in software. They are also expensive, in terms of silicon, to implement in hardware. As a result, very few embedded systems make full use of floating-point calculations.

Remember that RTA-OSEK generally assumes that floating-point is not used in an application. If an ISR does use floating-point, you must declare this in the RTA-OSEK GUI. RTA-OSEK Component will then ensure that any hardware registers that are involved with floating-point calculations are saved and restored on entry or exit for the ISRs.

RTA-OSEK is able to calculate exactly how much memory to reserve for saving the floating-point ISRs. It can do this because it can work out the worst-case preemption depth for ISRs that use floating-point. You can see the results of the calculation in the stack depth analysis of RTA-OSEK.

5.10 The Default Interrupt

If you are using RTA-OSEK to generate a vector table, then you may want to fill unused vector locations with a **default interrupt**.

Figure 5:7 shows how the default interrupt is defined.

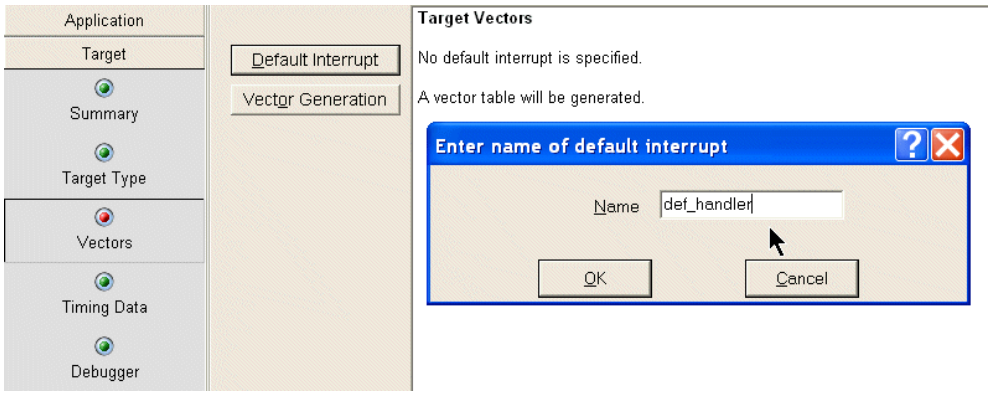


Figure 5:7 - Placing a Default Interrupt in the Vector Table

Portability: The default interrupt is not supported on all targets.

The default interrupt is slightly different to other interrupts. It is used to fill every location in the vector table for which you have not defined an interrupt. This feature has been provided as a debugging aid and as a means of providing a "fail-stop" in the event of erroneous generation of interrupts in production systems. If you actually want to attach interrupt handlers to vectors to do useful work, you should explicitly create them as ISRs.

There are limitations on the use of the default interrupt handler. It cannot make any OS calls, and system behavior is undefined if it ever returns.

Important: You must not make any RTA-OSEK Component API calls from the default interrupt and you must not return from the handler.

The last statement in your default interrupt handler should be an infinite loop. Code Example 5:5 shows how this can be done.

```
__interrupt void default_handler(void)
{
 /* invoke target-specific code to lock interrupts
*/
  asm("di"); /* or whatever on your platform */
  for (;;) {
  }
  /* Do NOT return from default handler. */
}
```

Code Example 5:5 - The Default Interrupt Handler

5.11 Shortening Interrupt Response Times

When you write a Category 2 interrupt handler it is better to make the handler as short as possible (especially on targets that support a single interrupt priority). Long running handlers will add additional latency to the servicing of lower priority interrupts.

By moving the required functionality to a task, a short interrupt handler can be written that simply activates the task and then terminates.

Code Example 5:6 and Code Example 5:7 show how these techniques differ.

```
#include "Interrupt1.h"
ISR(Interrupt1) {
  /* Long handler code. */
}
```

Code Example 5:6 - Long Interrupt Handler (Long Blocking)

```
#include "Interrupt1.h"

ISR(Interrupt1) {
  ActivateTask_Task1();
}
```

```
#include "Task1.h"

TASK(Task1) {
  /* Long handler code. */
  TerminateTask();
}
```

Code Example 5:7 - Short Interrupt Handler (Short Blocking)

Interrupts Summary

- Interrupts provide a mechanism to capture real-world stimuli.
- OSEK supports two categories of interrupts: Category 1 and Category 2.
- Category 1 interrupts are not processed by RTA-OSEK Component.
- Category 2 interrupts are processed by RTA-OSEK Component.

6 Resources

Access to hardware or data that needs to be shared between tasks and ISRs can be unreliable and unsafe. This is because task or ISR preemption can occur whilst a lower priority task or ISR is part way through updating the shared data. This situation is known as a **race condition** and is extremely difficult to test for.

You learnt earlier that a sequence of statements that accesses shared data is known as a **critical section**.

To provide safe access to code and data referenced in the critical section, **mutual exclusion** must be enforced. In other words, you must make sure that no other task or Category 2 ISR in the system is able to pre-empt the executing task during the critical section.

In an OSEK operating system, such as RTA-OSEK Component, mutual exclusion is provided by **resources**.

While a task or Category 2 ISR **gets** a resource, no other task or ISR can get the resource. When the critical section is finished, the task or ISR **releases** the resource.

Resources are like binary semaphores, but they are held according to a **locking protocol**. This locking protocol is called **priority ceiling protocol***.

Priority ceiling protocol uses the concept of a ceiling priority for the resource that is the highest priority of any task or ISR that gets the resource. When a task or ISR gets a resource, its priority is increased to the ceiling priority. When the resource is released, the priority reverts to the initial priority.

Priority ceiling protocol provides two major benefits:

- It removes priority inversion (which can occur with binary semaphores).
- It is guaranteed to be deadlock free.
  In addition to this you will see that each time a high priority task becomes *ready*, its execution can only be delayed by a single lower priority task that gets a resource. This results in a strict bound on the worst-case time that a task is delayed by other tasks with which it shares a resource.

6.1 Resource Configuration

RTA-OSEK needs to know which tasks and ISRs use which resources. It can then calculate the ceiling priorities used by the priority ceiling protocol.

Additional resource usage information for each task or ISR can be configured during task or ISR configuration. This information is needed for analysis only. You can declare up to 255 resources in your application.

* This is a variant of immediate inheritance

At the most basic level, resources only need to be named. Look at the example in Figure 6:1 to see how resources are configured in the RTA-OSEK GUI.

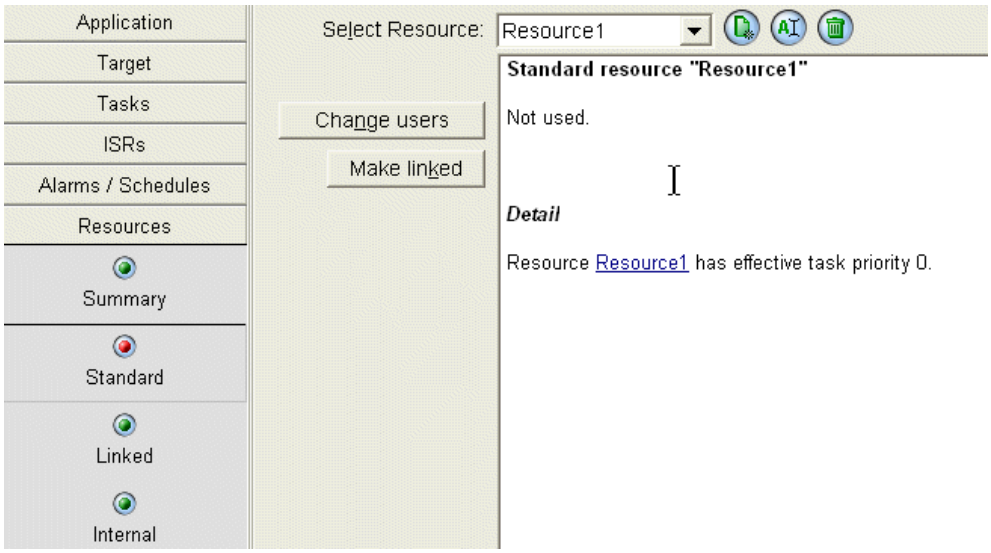


Figure 6:1 - Configuring Resources using the RTA-OSEK GUI

Figure 6:1 shows that a resource called `Resource1` has been declared. When you refer to this resource in your program you must use the same name.

6.2 Getting and Releasing Resources

You learnt earlier that when a task or Category 2 ISR gets a resource it prevents any other task or ISR from getting the resource.

You can get a resource using the `GetResource()` API call. You can then release a resource using the `ReleaseResource()` call. A task or ISR must not terminate until it has released all resources that are still held.

A task or ISR can only use the resources that you specify during RTA-OSEK Component configuration. Code Example 6:1 shows you how resources are used in `Task1`.

```
#include "Task1.h"

TASK(Task1) {

  /* Task functionality. */
  GetResource(Resource1);
  /* Critical section. */
  ReleaseResource(Resource1);
  /* Remainder of task functionality. */
  TerminateTask();
}
```

Code Example 6:1 - Using Resources

Important: Make sure that the `GetResource()` and `ReleaseResource()` calls are paired. In other words, you must not get a resource that is already held or release a resource that has not been held.

When a resource is held, it is effectively boosting the priority of the holding task or ISR to the resource priority. The resource priority is the highest priority of any task or ISR that shares the resource and is automatically calculated by RTA-OSEK.

In Figure 6:2, you can see that `Task1` has priority 3. This task shares a resource called `Resource1` with `Task2`. `Task2` has priority 7, so the resource priority will be 7 (the highest priority of all tasks that share the resource). When the resource is held, `Task1` runs at priority level 7, returning to priority level 3 when the resource is released.

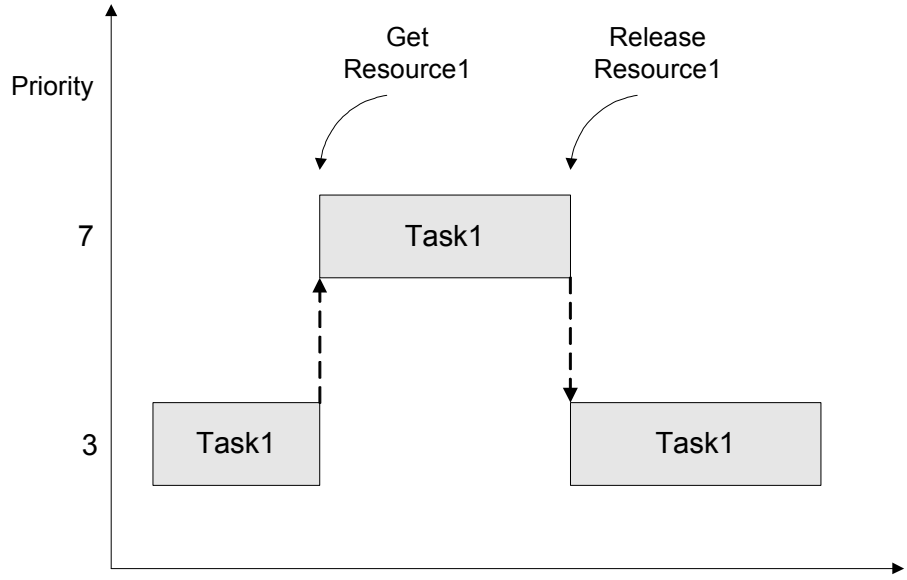


Figure 6:2 - Tasks Using Resources

6.2.1 Nesting Resource Calls

You can get more than one resource concurrently, but the API calls must be strictly nested. Let's look at two examples; one showing incorrectly nested calls and the other showing the API calls nested correctly. Code Example 6:2 shows `Resource1` and `Resource2` being released in the wrong order.

```
GetResource(Resource1);
GetResource(Resource2);
ReleaseResource(Resource1); /* Illegal – Resource2
ReleaseResource(Resource2);  * must be released
                             * first. */
```

Code Example 6:2 - Illegal Nesting of Resource Calls

A correctly nested example is shown in Code Example 6:3. All of the resources are held and then released in the correct order.

```
GetResource(Resource1);
GetResource(Resource2);
GetResource(Resource3);
ReleaseResource(Resource3);
ReleaseResource(Resource2);
ReleaseResource(Resource1);
```

Code Example 6:3 - Correctly Nested Resource Calls

6.3 Using the Static Interface

If a task does not state that it uses a given resource, it should not attempt to get the resource. This fact is enforced by the **static interface**.

The static interface is a mechanism used by RTA-OSEK to generate optimized system calls that can be used by your application. Static versions of the `GetResource()` and `ReleaseResource()` API calls are provided.

Look at the following two examples where `Resource1` is held and then released. You can see that Code Example 6:4 uses dynamic calls. Compare this with Code Example 6:5, which uses the static calls.

```
GetResource(Resource1);     /* Dynamic call. */
/* Critical section. */
ReleaseResource(Resource1);
```

Code Example 6:4 - Dynamic Resource Calls

```
GetResource_Resource1();     /* Static call. */
/* Critical section. */
ReleaseResource_Resource1();
```

Code Example 6:5 - Static Resource Calls

For optimum performance, you should use the static versions of the calls wherever possible. You will only need to use a dynamic call when a resource is unknown at compile time (for example, if it is passed as a parameter to a function in a library).

Using the static version allows RTA-OSEK to calculate whether or not any action needs to be taken. So, for instance, the highest priority task or ISR that locks a resource does not need to do anything. This is because the priority will already match the resource level. The `GetResource()` and `ReleaseResource()` calls are mapped to an empty statement.

6.4 Combined Resources

Resources that are shared between tasks and interrupts are called **combined resources**.

RTA-OSEK will automatically identify the resources that are combined resources, so you don't need to do any special configuration.

When a combined resource is held by a task, getting the resource will mask all interrupts with interrupt priority less than or equal to the highest priority interrupt that shares the resource.

This is simply an extension of priority ceiling protocol that supports combined resources. This behavior means that it is possible to mask individual interrupts at a particular priority level.

6.5 Linked Resources

In OSEK, `GetResource()` API calls for the same resource cannot be nested. However sometimes, there are cases where you may need to nest the calls.

Your application may, for instance, use a function shared amongst a number of tasks. What happens if the shared function needs to get a resource used by one of the tasks, but not by the others? Have a look at Code Example 6:6.

```
#include "Task1.h"
TASK(Task1) {
  …
  GetResource(Resource1);
  /* Critical section. */
  SomeFunction();
  ReleaseResource(Resource1);
  …
}
```

```
#include "Task2.h"
TASK(Task2) {
  …
  SomeFunction();
  …
}
```

```
#include "osek.h" /* Generic header file. */
void SomeFunction(void) {
  …
  GetResource(Resource1);      /* Not allowed! */
  /* Critical section. */
  ReleaseResource(Resource1);  /* Not allowed! */
  …
}
```

Code Example 6:6 - Illegally Nested Resource API Calls

In these cases, the nesting of a (potentially) held resource must use **linked resources**. A linked resource is an alias for an existing resource and protects the same, shared, object.

Figure 6:3 shows how linked resources are declared using the RTA-OSEK GUI.

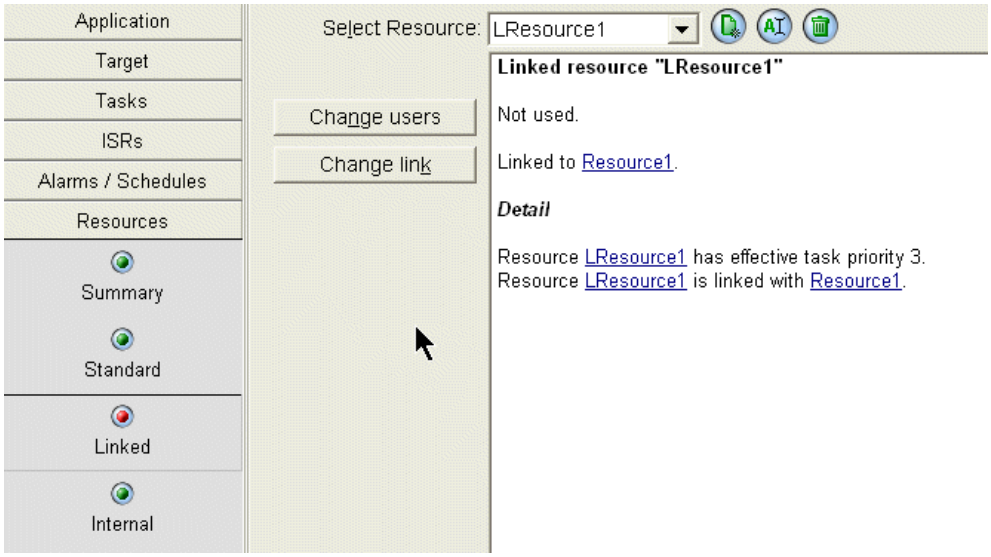


Figure 6:3 - Configuring a Linked Resource in the RTA-OSEK GUI

Linked resources are held and released using the same API calls for standard resources (these are explained in Section 6.2). You can also create linked resources to existing linked resources.

6.6 Internal Resources

If a set of tasks share data very closely, it becomes too difficult for resources to guard each access to each item of data. You may not even be able to identify the places where resources need to be held.

You can prevent concurrent access to shared data by using **internal resources**. Internal resources are resources that are allocated for the lifecycle of a task.

Internal resources are configured offline using the RTA-OSEK GUI. Unlike normal resources, however, you cannot get and release them and they are not available to ISRs.

Internal resources in RTA-OSEK Component do not consume any processor resources at run-time because RTA-OSEK performs calculations during the build process.

The set of tasks that share an internal resource is defined at configuration time using the RTA-OSEK GUI. This membership is static.

Figure 6:4 shows the declaration of an internal resource, called `IntResource1`, which is shared between two tasks called `t1` and `t3`.

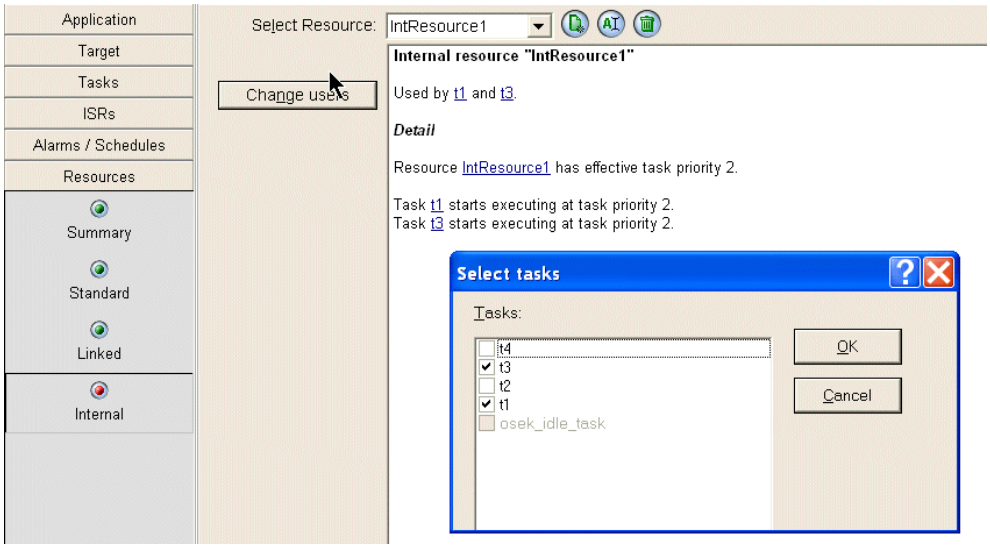


Figure 6:4 - Declaring an Internal Resource using the RTA-OSEK GUI

If a task uses an internal resource, RTA-OSEK Component will automatically get the specified resource before calling the task's entry function. The resource will then be automatically released after the task terminates, makes a `Schedule()` call or makes a `WaitEvent()` call.

During task execution, all other tasks sharing the internal resource will be prevented from running until the internal resource is released. Preemption, however, is still possible by all higher priority tasks that do not share the internal resource. You can see an illustration of this in Figure 6:5.

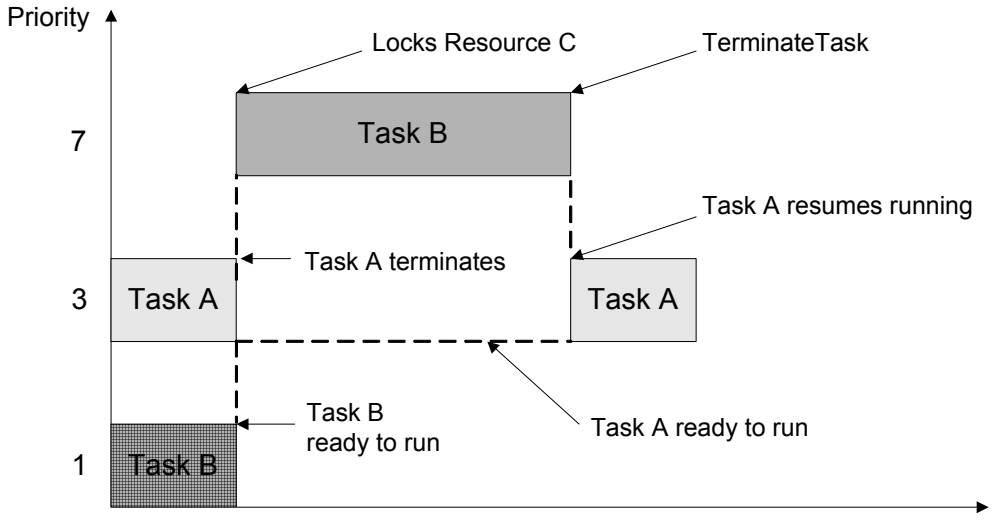


Figure 6:5 - Preemption of Tasks that do not Share an Internal Resource

Figure 6:5 shows that Task A is *running* and Task B is *ready*, but Task B has a lower priority. When Task A terminates, Task B runs because it shares an internal resource with Task C and the resource has a priority level of 7.

Task A is *ready* to run, but cannot preempt Task B because Task B still gets the resource with priority level 7. When Task B terminates, Task A resumes *running*.

Tasks that share an internal resource run non-preemptively with respect to each other. You saw earlier that non-preemptive tasks can be used, but remember that they run non-preemptively with respect to the entire application.

Using internal resources gives you greater control over the timing behavior of your application. Internal resources are also useful for reducing the memory used by your system by limiting the total amount of preemption.

Tasks that share an internal resource will run sequentially, but only one of the tasks will be held on the stack at any given time. As a result, the overall stack space required is reduced.

Advanced Resource Concepts

6.7 Using the Scheduler as a Resource

A task can hold the scheduler if it has a critical section that must be executed without pre-emption from any other task in the system (remember that the scheduler is used to perform task switching). A predefined resource called `RES_SCHEDULER` is available to all tasks for this purpose.

For tasks, getting `RES_SCHEDULER` is equivalent to declaring the task to be non-preemptive. However, getting `RES_SCHEDULER` is better than using non-preemptive tasks, particularly when a task only needs to prevent pre-emption for a short part of its total execution time.

Using `RES_SCHEDULER` can improve response times of the tasks that might otherwise suffer multiple preemption from other tasks in the application. This can result in a long response time for the task. If you do not use `RES_SCHEDULER` you can disable this in the Application Optimizations. This results in a time and space saving.

6.8 Choosing a Pre-Emption Control Mechanism

If code that does not require locks appears between a pair of `GetResource()` and `ReleaseResource()` calls, the system responsiveness can potentially be reduced.

With this in mind, when you use resources in your application, you should place get calls as closely as possible around the section of code you are protecting with the resource.

However, there is an exception to this rule. This exception occurs when you have a short running task or ISR that makes many `GetResource()` and `ReleaseResource()` calls to the same resource. The cost of the API calls may then make up a significant part of the overall task execution time and, therefore, potentially the response time.

You may find that placing the entire task or ISR body between `GetResource()` and `ReleaseResource()` calls actually shortens the worst-case response time.

You should avoid using non-preemptive tasks and getting `RES_SCHEDULER` wherever possible. System responsiveness and schedulability is improved when resources are held for the minimum amount of time and when this affects the smallest number of tasks.

6.9 Avoiding Race Conditions

The OSEK standard specifies that resources must be released before a `TerminateTask()` call is made. In some circumstances, this can introduce a race condition into your application. This can cause task activations to be missed (you learnt about race conditions at the beginning of this chapter).

Code Example 6:7 shows the type of system where race conditions can become a problem. Assume that two BCC1 tasks exchange data over a bounded buffer.

```
#include "Write.h"

TASK(Write) {     /* Highest priority .*/

  WriteBuffer();
  GetResource(Guard);
  BufferNotEmpty = True;
  ReleaseResource(Guard);
  ChainTask(Read);

}

#include "Read.h"

TASK(Read) {      /* Lowest priority. */

  ReadBuffer();
  GetResource(Guard);

  if( BufferNotEmpty ) {
    ReleaseResource(Guard);
    /* Race condition occurs here. */
    ChainTask(Read);
  } else {
    ReleaseResource(Guard);
    /* Race condition occurs here. */
    TerminateTask();
  }

}
```

Code Example 6:7 - A System where a Race Condition can Occur

In Code Example 6:7, between the resource being released and the task terminating, `Read` can be pre-empted by `Write`. When task `Write` chains

task `Read` the activation will be lost. This is because `Read` is still *running*. In other words a task is being activated, but it is not in the *suspended* state.

To solve this problem, you can allow queued activations of the `Read` task. This means that you should make the task BCC2.

Resources Summary

- Resources are used to provide mutual exclusion over access to shared data or hardware resources.
- Tasks and ISRs can share any number of resources.
- All `GetResource()` and `ReleaseResource()` calls must be properly nested.
- All resources must be released before the task or ISR terminates.
- The scheduler can be released as a resource, but internal resources should be used in preference, if possible.
- Internal resources provide a cost free mechanism for controlling preemption between a group of tasks.

7 Events

In an OSEK system, events are used to send signal information to tasks. You will learn how to configure events in Section 7.1.

Events can be used to provide multiple **synchronization points** for extended tasks. A visualization of synchronization is shown in Figure 7:1.

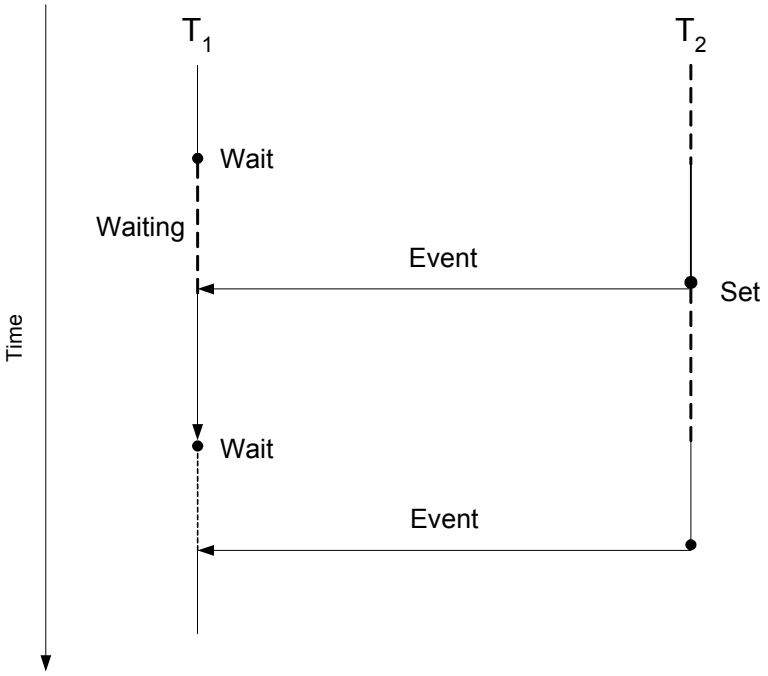


Figure 7:1 - Visualizing Synchronization

An extended task can **wait** on an event, causing the task to move into the *waiting* state. You'll learn more about this in Section 7.2.

When an event is **set** by a task or ISR in the system, the *waiting* task is transferred into the *ready* state. When it becomes the highest priority task it will be selected to run by RTA-OSEK Component.

Events are owned by the extended task with which they are associated. Usually an extended task will be an infinite loop that contains a series of guarded wait calls for the events it owns.

If timing behavior is important in your system, all of your extended tasks (in other words, any task that declares an event) must be lower priority than the basic tasks.

7.1 Configuring Events

Events are declared using the RTA-OSEK GUI. The maximum number of events that can exist in your application is determined by your target hardware. You can see the maximum number of events by looking at the Target Summary in the RTA-OSEK GUI.

In Figure 7:2, the target can wait on a maximum of 16 events.

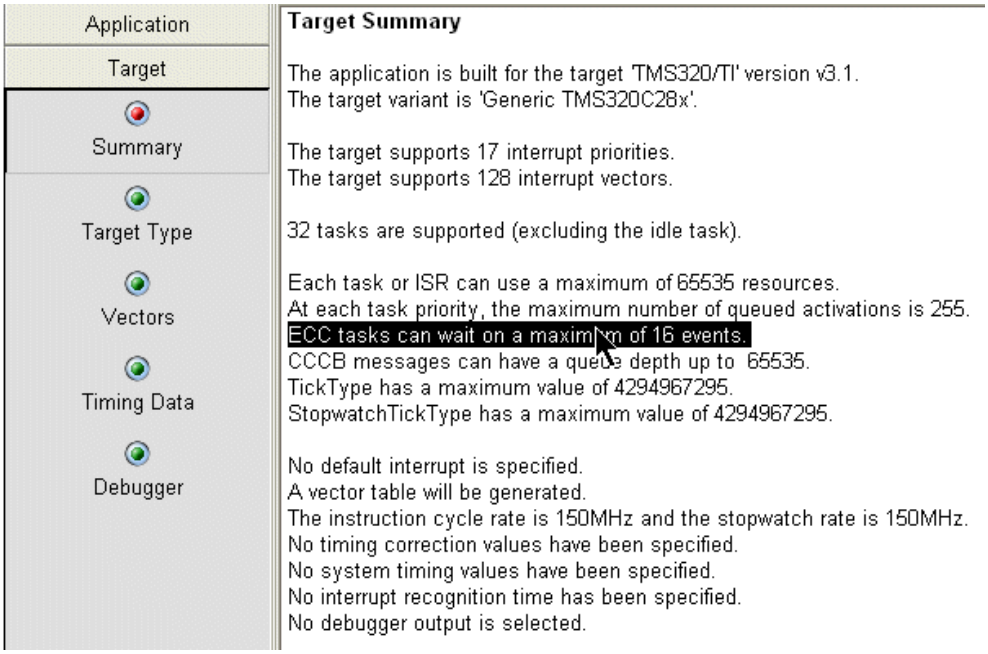

Application
Target
Summary
Target Type
Vectors
Timing Data
Debugger
Target Summary
The application is built for the target 'TMS320/TI' version v3.1.
The target variant is 'Generic TMS320C28x'.
The target supports 17 interrupt priorities.
The target supports 128 interrupt vectors.
32 tasks are supported (excluding the idle task).
Each task or ISR can use a maximum of 65535 resources.
At each task priority, the maximum number of queued activations is 255.
ECC tasks can wait on a maximum of 16 events.
CCCB messages can have a queue depth up to 65535.
TickType has a maximum value of 4294967295.
StopwatchTickType has a maximum value of 4294967295.
No default interrupt is specified.
A vector table will be generated.
The instruction cycle rate is 150MHz and the stopwatch rate is 150MHz.
No timing correction values have been specified.
No system timing values have been specified.
No interrupt recognition time has been specified.
No debugger output is selected.

Figure 7:2 - Viewing the Maximum Number of Events for a Target

When an event is declared it must have:

- A name.
  Names are used only to indicate the purpose of an event at configuration time.
- At least one task that uses it.
- An event mask.
  The event name that is specified in the RTA-OSEK GUI is used as a symbolic name for the event mask at run-time. A mask is an `N` bit vector with a single bit set, where `N` is the maximum number of events on which a task can wait. The set bit identifies a particular event.

The event name is used at run-time as a symbolic name for the mask. The mask can be declared explicitly or, alternatively, RTA-OSEK can generate the mask automatically for you. When several tasks wait on many events, it is better to allow RTA-OSEK to generate the mask automatically.

Figure 7:3 shows that an event called `Event1` has been declared. In this example, you can see that RTA-OSEK will automatically generate the event mask and the event is used by a task called `t3`.

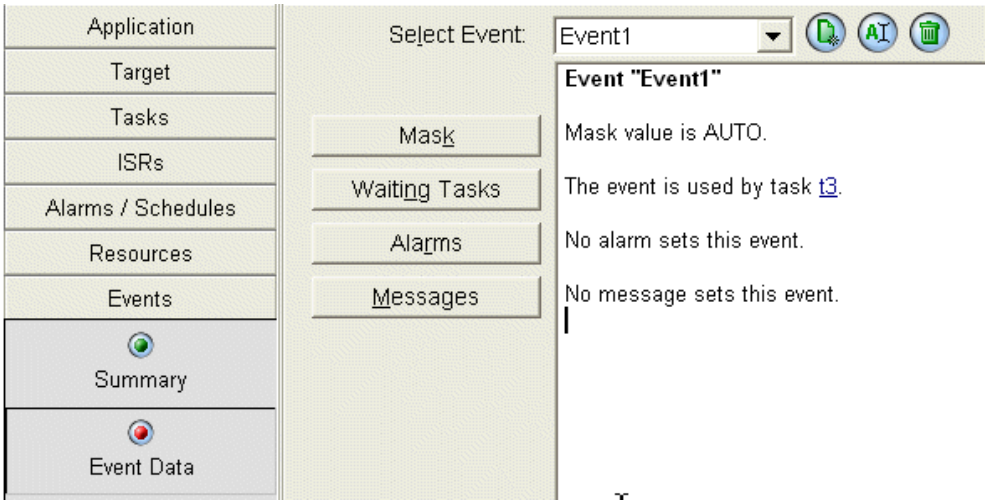


Figure 7:3 - Declaring an Event in the RTA-OSEK GUI

If an event is used by more than one task, each task has its own individual copy. When an event is set, a task must be specified at the same time. So, for example, if you set an event called `Event2` for a task called `t3`, this has no effect on `Event2` for the task `t4`. When a task terminates all the events that it owns are cleared.

7.2 Waiting on an Event

Waiting tasks are selected using the RTA-OSEK GUI. If you declare a task that waits on an event, it automatically means that it will be treated as an extended task.

Figure 7:4 shows you that the event, called `Event1`, has been declared and that the tasks t2 and `t3` are being configured to wait on the event.

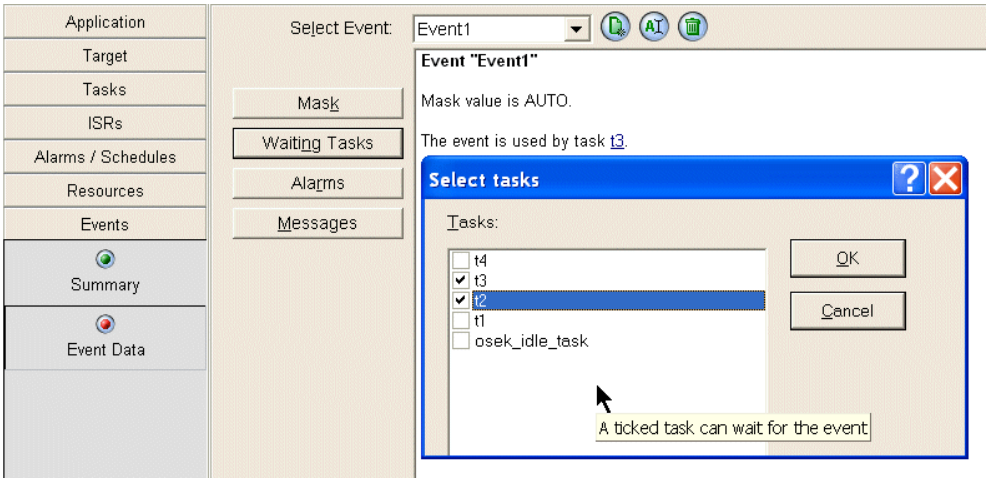


Figure 7:4 - Selecting a Task to Wait on an Event

An extended task that waits on an event will usually be autostarted and the task will never terminate. When the task starts executing, all the events it owns are cleared by RTA-OSEK Component.

If the task is to wait for an event, it will issue the `WaitEvent(EventMask)` API call. The `EventMask` must correspond to the one that is declared in the RTA-OSEK GUI.

Using this call causes the task to move from the *running* state into the *waiting* state. If, however, the event has already occurred the task remains in the *running* state.

7.3 Setting Events

Events are set using the `SetEvent()` API call.

The `SetEvent()` call has two parameters, a task and an event mask. For the specified task, each call sets the events that are specified in the event mask. The call does not set the events for any other tasks that share the events.

Events cannot be set for tasks that are in the *suspended* state*. So, before setting the event, you must be sure that the task is not *suspended*.

An extended task is moved from the *waiting* state into the *ready* state when any one of the events that it is waiting on is set.

Code Example 7:1 shows you how a task can set events.

```
TASK(Task1) {
  …
  SetEvent(Task2, Event1);
  SetEvent(Task3, Event1);
  …
  TerminateTask();
}
```

Code Example 7:1 - Setting Events

A number of tasks can wait on a single event. However you can see from Code Example 7:1 that there is no broadcast mechanism for events. In other words, you cannot signal the occurrence of an event to all tasks waiting on the event with a single API call. If you do want to do this, then RTA-OSEK tasksets can provide similar functionality.

7.4 Clearing Events

An event can be set by any task or ISR, but only the owner of the event can clear it.

You can clear events using the `ClearEvent(EventMask)` API call. The `EventMask` must correspond to the one that is declared in the RTA-OSEK GUI. Before you return to `WaitEvent()`, you must clear the event that you are responding to.

* This implies that the body of extended tasks should be an infinite loop that waits on events.

7.5 Events and the Idle Task

You have seen that the idle task is the lowest priority task in the system. In RTA-OSEK Component the idle task can wait on events, which means that it can be an extended task.

Unlike other extended tasks, an extended idle task will never need to be moved off the stack when it issues a `WaitEvent()` call. RTA-OSEK, therefore, does not need to allocate memory to save the current context.

These facts provide a useful optimization. If you need a single extended task in your application you can use the idle task, without any time or space penalty being applied to the rest of your application.

A system with the idle task as the only extended task has exactly the same performance as strict basic conformance class systems. In practice, this means that the idle task can be turned into an extended task at no cost.

The idle task can be used if you want the system to remain BCC for timing analysis, but would like to use a single extended task. If you use this method you will avoid compromising the timing behavior of the rest of the system.

7.6 Waiting On, Setting and Clearing Multiple Events

The event API calls take a bit mask parameter to indicate which event to wait on, to set or to clear.

You can wait on, set or clear multiple events with a single API call by bitwise 'OR'ing a set of bit masks. Have a look at Code Example 7:2 to see how a task can wait on multiple events simultaneously.

```
TASK(Task1){
  EventMaskType eventMask;

  /* Wait on three specified events. */

  while( WaitEvent(Event1|Event2|Event3)==E_OK ) {
    GetEvent(Task1, &eventMask);
    if( eventMask & Event1 ) {
      ClearEvent(Event1);
      ...
    } else if( eventMask & Event2 ) {
      ClearEvent(Event2);
      ...
    } else if( eventMask & Event3 ) {
      ClearEvent(Event3);
      ...
    }
  }
}
```

Code Example 7:2 - Waiting on Multiple Events

7.7 Advanced Event Setting

In Section 7.3 you saw how tasks and ISRs can use the `SetEvent()` API call to indicate that an event has occurred.

There are, in fact, other methods you can use to do this. Events can be set with alarms and with messages.

7.7.1 Setting Events with an Alarm

Alarms can be used to periodically activate extended tasks that don't terminate. Each time the alarm expires, the event is set. The task *waiting* on the event is then made *ready* to run.

7.7.2 Setting Events with a Message

COM messages can be configured to set an event on transmission. Each time the message is sent the event is set. The task *waiting* on the event will be made *ready* to run.

Events Summary

- Events are synchronization objects that can be waited on by extended tasks.
- An event is owned by a single task.
- Tasks, ISRs, alarms and messages can set events.

8 Messages

Tasks and interrupts will often need to communicate. For example, a communication bus interrupt may want to pass information to a task telling it how many bytes to read from a shared buffer.

Communication between objects can be achieved using **message passing**. In RTA-OSEK, message passing is **asynchronous**. This means that when the message is sent, the sender continues to execute. When the receiver begins to execute, it consumes the sent message.

All data transmission is memory to memory because messages are only sent between objects on a single CPU. There is no concept of a transmission failure.

8.1 Communication in OSEK

Message passing in an OSEK operating system is defined by a subset of the OSEK Communications (COM) standard.

In RTA-OSEK, message passing satisfies the OSEK COM conformance class CCCB for internal task and interrupt communication.

OSEK COM CCCB provides facilities for communication between tasks and ISRs. CCCB supports both queued and non-queued message transmission.

8.2 Configuring Messages

If you want to use the communication facilities in RTA-OSEK Component you'll need to declare a COM message in the RTA-OSEK GUI.

There are a number of stages involved in configuring a COM message:

- Declare the message.
- Declare the sender and receiver(s).
- Specify the accessors.
- Specify the transmission mechanism.

Each of these stages will now be explained in more detail.

8.2.1 Declaring Messages

Messages are declared using the RTA-OSEK GUI. You can see, from Figure 8:1, how a new message has been added to the application.

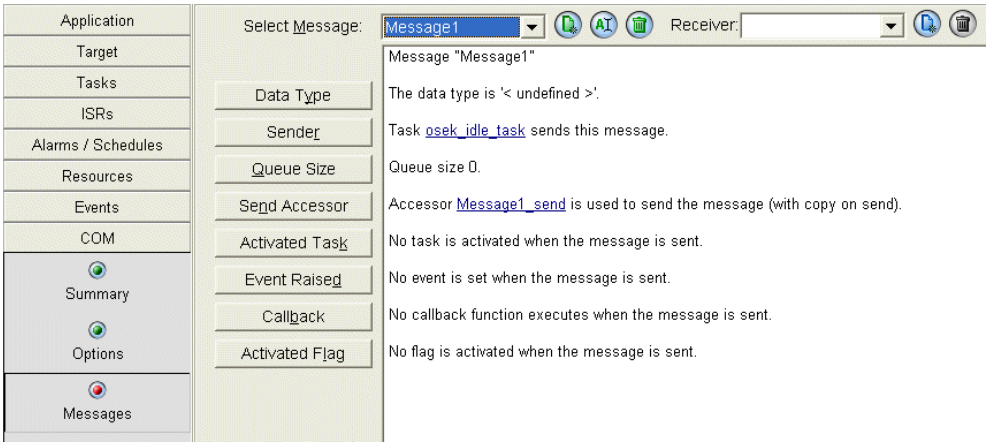


Figure 8:1 - Declaring a New Message

At the simplest level, each message must have:

- A name.
  The name is used to refer to the message at run-time.
- A data type.
  The data type specifies the content of the message.
  This is the C type of the actual message data. This could be a simple type, such as `unsigned char`, or it could be a more complex type, such as `struct myMessage`.

Figure 8:2 shows a message called `Message1` that uses an integer data type.

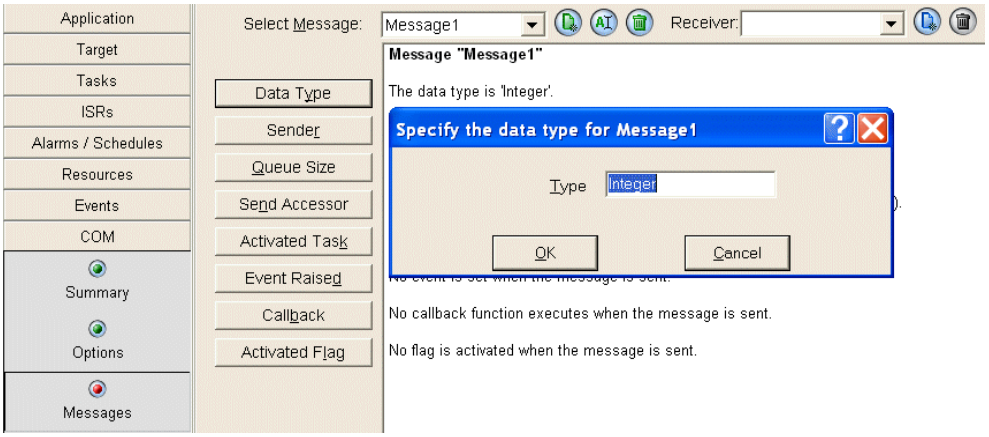


Figure 8:2 - Specifying the Data Type for a Message

COM does not need to know the type or even the content of a message. However, this information is needed by RTA-OSEK at build time, so that the correct amount of memory can be allocated for messages during the build process.

Any type of data, such as integers, arrays, strings and linked lists can be passed as a message. If the data type isn't a standard C or standard RTA-OSEK data type, it must be declared in a file.

By default, the information is located in the file called **`comstrct.h`**. You can, however, choose a different file for the data type definitions.

The name of this file is specified in the RTA-OSEK GUI using the COM Options. In Figure 8:3 the existing file is called `comstrct.h`. The file can be renamed by typing in a new name. If you do not want a `#include` file, delete the name and leave it blank.

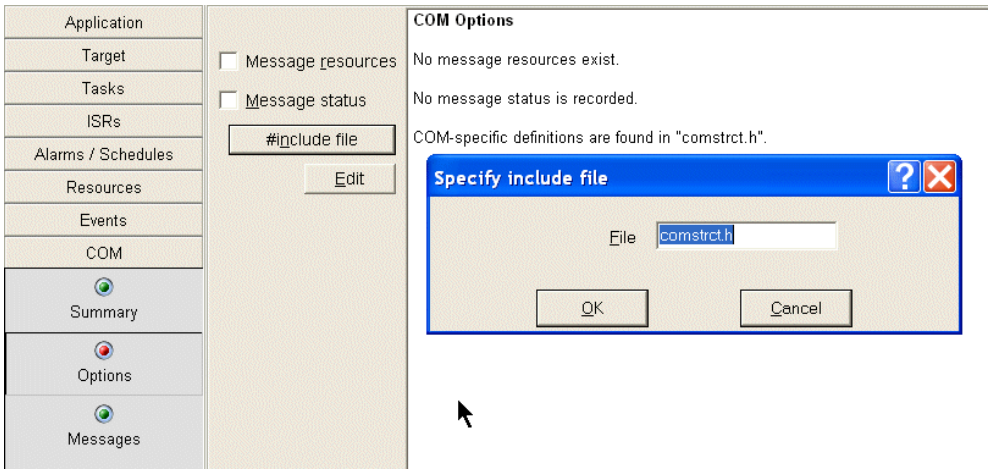


Figure 8:3 - Specifying the Name of the Include File

The type specified for a COM message must be a complete C language type – i.e. the type must be something that could be used for declaring a variable. (Amongst other things RTA-OSEK uses the specified type to create message accessors.) For example, assume that the following declarations appear in **`comstruct.h`**:

```
struct myStruct
{
  char a;
  char b;
};

typedef struct
{
  char x;
  char y;
}
myRecord;

typedef char myArray[10];
```

The following would be valid message types:

```
int
struct myStruct
myRecord
myArray
```

8.2.2 Declaring Senders and Receivers

A message is sent by a single sender, but it can be received by multiple receivers. This provides a mechanism for broadcasting information to the whole system.

The sender and receiver of a particular message must be specified at configuration time using the RTA-OSEK GUI. Both tasks and Category 2 ISRs can be senders or receivers.

Figure 8:4 shows you how the sender, `Task1`, is specified for `Message1`. Notice how only one sender can be specified for each message.

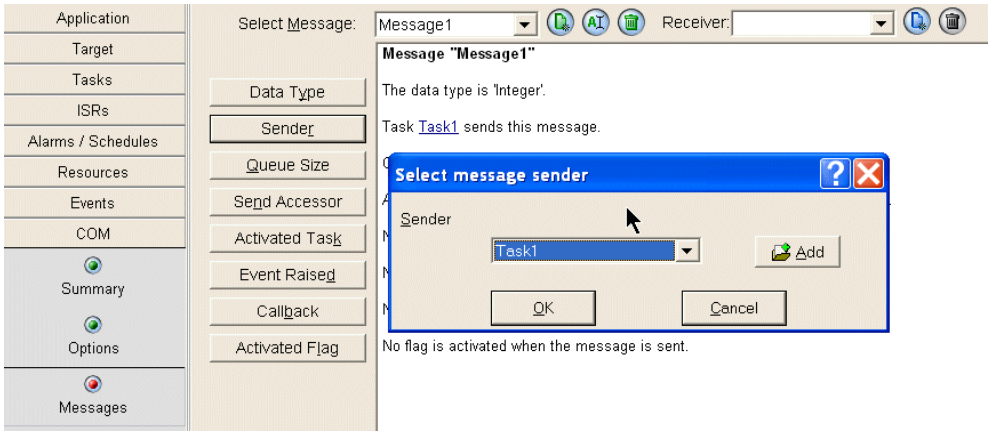


Figure 8:4 - Declaring a Sender for a Message

In Figure 8:5, a Receiver called `Task2` has been added to `Message1`. Each message can have any number of receivers added in this way.

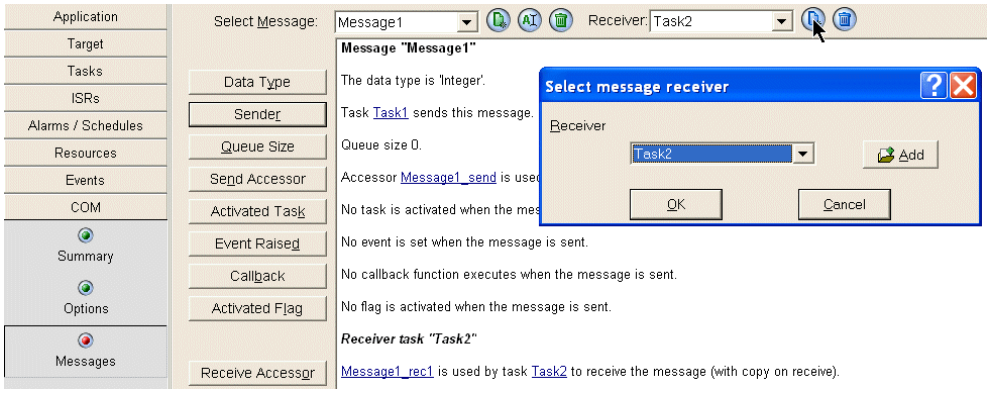


Figure 8:5 - Adding Receivers to a Message

8.2.3 Specifying Accessors

Senders and receivers manipulate message data using **accessors**. Accessors are used by the application to send and receive message data using the corresponding API calls.

Accessors must be declared for both the sender and receiver. They are also unique to a task or ISR message pairing.

An accessor is a reference to a data object with the same type as the message. Depending on the message characteristics, an accessor can reference either the actual data in the message or a copy of it.

Figure 8:6 shows how the RTA-OSEK GUI is used to create a send accessor called `Message1_send`.

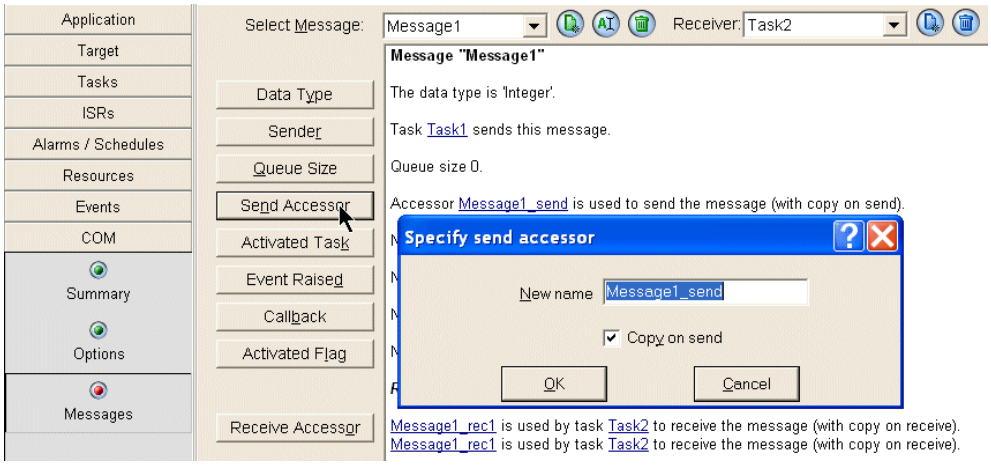


Figure 8:6 - Specifying a Send Accessor

The diagram in Figure 8:7 shows how accessors are used to pass messages between tasks or ISRs.

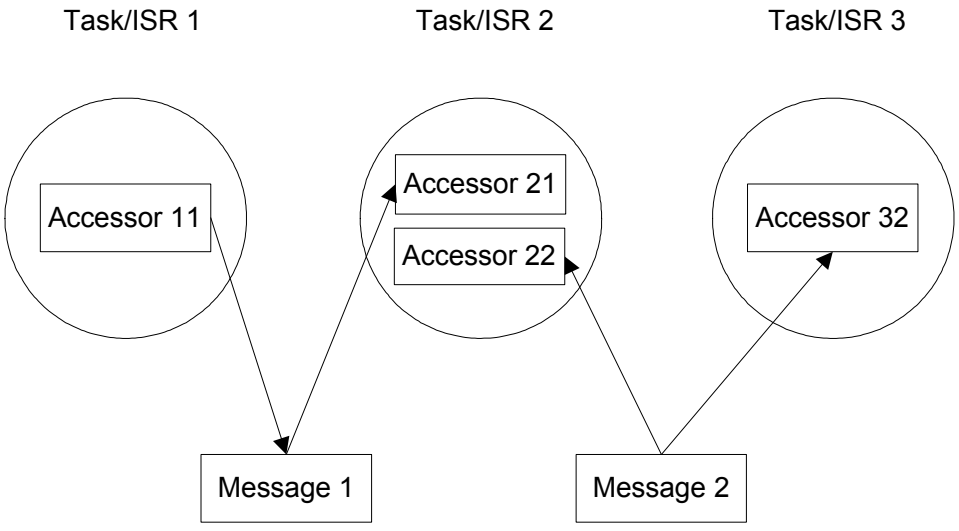


Figure 8:7 - Using Accessors to Send and Receive Message Data

If tasks or ISRs want to send or receive the same message they must use different accessors. RTA-OSEK creates an accessor named `<MessageID>_send` for the sender and an accessor named `<MessageID>_recN` for the receiver where `N` is the number of the receiver. Accessor names are unique to each task and ISR.

These symbolic names for the accessors can be changed, an example is shown in Figure 8:8. Here `Message1` has an accessor called `Message1_rec1`. In this example, the accessor is being renamed to `Rec1Message1`.

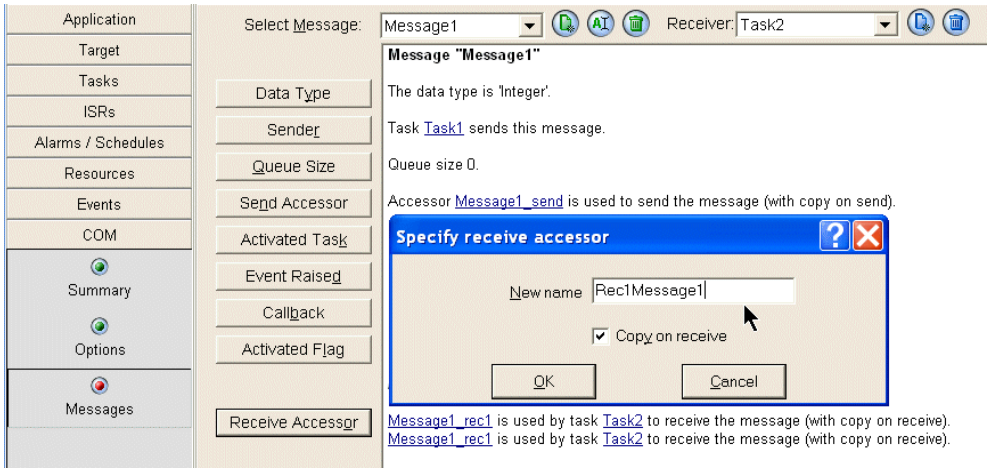


Figure 8:8 - Specifying a Receive Accessor

Accessors are used as if they were a variable of the message type (actually they are). So continuing the example in section 8.2.1, if we had a message called `message1` of type `myArray` with a send accessor called `message1_send` and a message called `message2` of type `struct myStruct` with a send accessor called `message2_send` then the following uses of the accessors would be legal:

```
for (i = 0; i < 10; i++)
{
  message1_send[i] = (char) i;
}

message2_send.a = 1;
message2_send.b = 2;
```

Important: You must make sure that, when passing the accessor into an API call, you pass the address of the accessor (this means that all calls should use `&AccessorName`).

8.2.4 Specifying Transmission Mechanisms

Accessors provide access to the message area, but they do not specify how messages are sent. OSEK COM defines two message transmission mechanisms:

- `WithCopy`.
- `WithoutCopy`.

WithCopy Mode

In the **WithCopy** transmission mechanism, the accessor references a local copy of the message. When a message is sent, RTA-OSEK Component copies the contents of the local copy into the message buffer.

When the message is received, the contents of the message location are copied to the local copy area for the receiver.

`WithCopy` mode can be used by both tasks and ISRs.

Important: `WithCopy` is the only transmission mode available to ISRs.

Have a look at the example in Figure 8:9 where both accessors are declared as `WithCopy`. (In the example, the big arrows indicate the copying of the message data and the small arrows indicate references.)

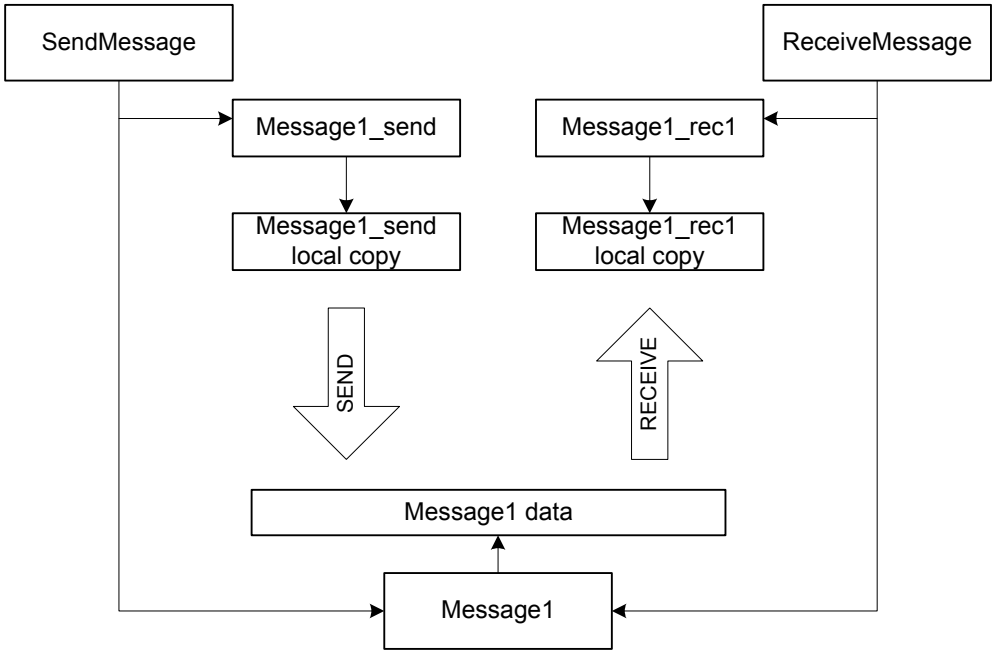


Figure 8:9 - Sending and Receiving Messages using WithCopy

This mechanism can be expensive if the message types are large and/or if they are complex data structures. However, `WithCopy` allows the sender and receiver to manipulate the copy of the message without affecting the message itself.

WithoutCopy Mode

Tasks can specify message transmission **WithoutCopy** (remember that ISRs cannot use this mode).

Using `WithoutCopy`, the accessor directly references the data buffer of the message, which saves the expensive copy operation that you saw in the previous example.

So, let's compare Figure 8:9 with our next example, Figure 8:10. Figure 8:10 illustrates the message transfer where both accessors are declared

as `WithoutCopy` (remember that in the diagram, the small arrows indicate references).

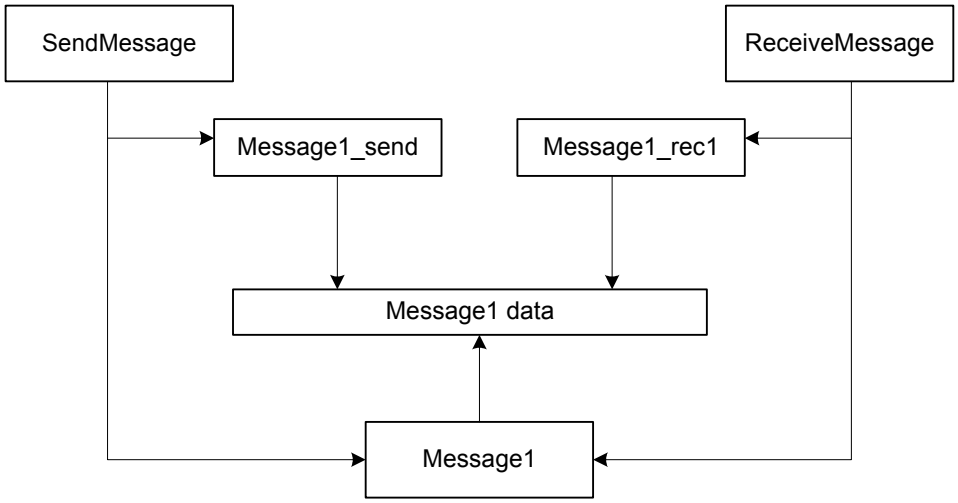


Figure 8:10 - Sending and Receiving Messages WithoutCopy

In `WithoutCopy` mode many tasks can have access to the same data area at the same time. It is up to you to make sure that no access conflicts occur (possibly by providing concurrency control over reads and writes to the message).

You can use the RTA-OSEK GUI to create a **message resource** for each message. A message resource is similar to a standard OSEK resource, but is specific to a particular message.

The message resource has the same name as its message. It is also available to each task and ISR that can access the message. When you get a message resource, the system's active priority should be raised to that of the highest priority task or ISR that can access the message.

The resource is accessed by the `GetMessageResource(MessageID)` and `ReleaseMessageResource(MessageID)` API calls. You can find out more information about these calls in the *RTA-OSEK Reference Guide*.

8.3 Sending and Receiving Messages

A message is sent using the `SendMessage(MessageID, &AccessorID)` call.

A task or ISR that wants to send a message must have a send accessor for that message. This must be declared in the RTA-OSEK GUI. Note that it is an error to call this API with an invalid message or accessor.

8.3.1 Sending a Message

To send a message you must:

- Copy the data that needs to be sent into the data buffer that `AccessorID` is pointing to. The accessor that you use must be valid for the task or ISR.
- Call the `SendMessage(Message, &Accessor)` API call. `Message` is the identifier of a declared message and `Accessor` is a reference to an accessor that the task or ISR is allowed to use.

In Code Example 8:1, the message `DataArrived` is sent by an ISR. The message type is defined by the `struct MyMessage`.

```
struct MyMessage {
  char  text[6];
  bool  aFlag;
};
ISR (MessageArrived) {

  /* Prepare data for sending. */
  DataArrived_send.aFlag = true;
  memcpy(DataArrived_send.text, "HELLO", 6);

  /* Send the message. */
  SendMessage(DataArrived, &DataArrived_send);

}
```

Code Example 8:1 - Sending a Message from an ISR

8.3.2 Receiving a Message

The `ReceiveMessage(MessageID, &AccessorID)` API call is used to receive messages. A task or ISR that wants to receive a message must have a receive accessor for that message declared in the RTA-OSEK GUI.

To receive a message you must:

- Call the `ReceiveMessage(Message, &Accessor)` API call. `Message` is the identifier of a declared message and `Accessor` is a reference to an accessor that the task or ISR is allowed to use.
- After the call has returned, access the data from the data buffer that `Accessor` is pointing to.

Code Example 8:2 shows the task `ProcessData` receiving a message.

```
TASK(ProcessData) {

  char buffer [6];

  /* Receive the message. */
  ReceiveMessage(DataArrived, &DataArrived_Rec1);
```

```
  /* Retrieve data from accessor. */
  memcpy(buffer, DataArrived_Rec1.text, 6);

}
```

Code Example 8:2 - A Task Receiving a Message

8.4 Starting and Stopping COM

The `StartCOM()` API call must be called before sending or receiving messages. This call initializes the implementation specific internal states and variables.

You can use `StopCOM(COM_SHUTDOWN_IMMEDIATE)` to stop COM at any time. In extended error checking mode this will cause subsequent send and receive operations to return the status `E_COM_SYS_STOPPED`.

8.5 Initialization and Shutdown of COM

API calls are provided to initialize and shutdown COM. These calls are intended for use where external hardware, such as a CAN driver, is used to pass messages to other processors. This is not directly supported in RTA-OSEK Component; however, you can do this using other add-on libraries.

The `InitCOM()` call can be used to initialize the network hardware. This should be called before `StartCOM()` and it is usually called from the startup hook.

The `CloseCOM()` API call is used to deactivate the network hardware. It should be called after `StopCOM()` and it is normally called from the shutdown hook.

Code Example 8:3 shows you how COM can be initialized and shutdown from the `StartupHook()` and `ShutdownHook()`.

```
OS_HOOK(void) StartupHook(void) {
  InitCOM;
}

OS_HOOK(void) ShutdownHook(StatusType status) {
  CloseCOM();
}
```

Code Example 8:3 - Hook Routines for Starting and Shutting Down COM

The `MessageInit()` callback function can be used to initialize user message objects. This is a user provided function that is called automatically from `StartOS()`. By default, RTA-OSEK Component provides this function automatically; however it can be overwritten by a user provided function, shown in Code Example 8:4.

```
StatusType MessageInit(void)
```

Code Example 8:4 - MessageInit() Callback Function

The status type returned by `MessageInit()` will be passed back as the `StartCOM()` API call status code.

Advanced Message Concepts

8.6 Queued Messages

The messages you have seen so far have been non-queued. RTA-OSEK Component supports COM Conformance Class B (CCCB) that provides facilities for queued message transmission.

For queued messages, RTA-OSEK Component maintains an internal FIFO (first-in, first-out) queue. You must specify the size of the queue when configuring your message in the RTA-OSEK GUI. When you do this, RTA-OSEK then knows how much space it needs to allocate. Figure 8:11 shows how the queue size is specified.

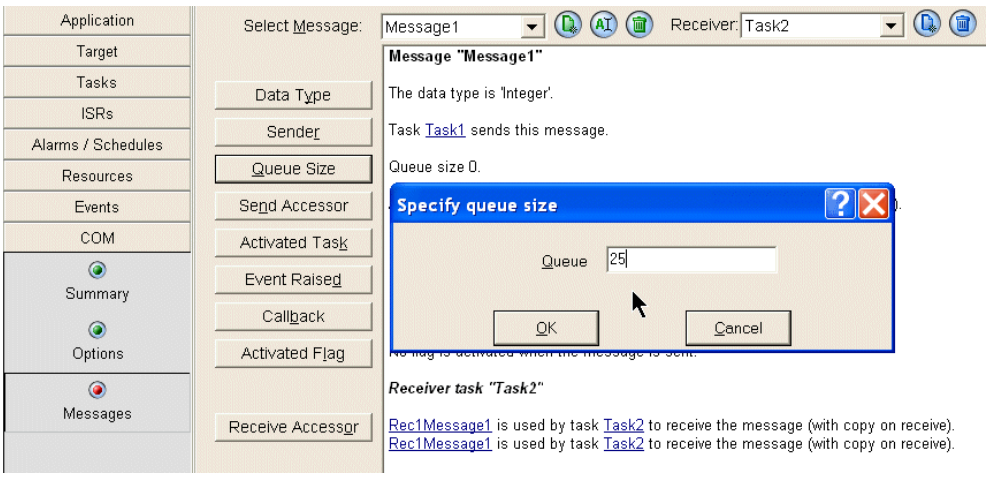


Figure 8:11 - Specifying the Queue Size

Queued messages have the same message names and accessors as non-queued messages, but there are important differences.

For queued messages:

- The transmission mode must always be `WithCopy` for both the sender and the receiver.
- Only a single receiver can be declared for each queued message. This is because queued messages have **destructive read**. So, when a receiver reads the message at the head of the queue, that message is then deleted.
- ISRs cannot send queued messages.

8.7 Mixed-Mode Transmission

In Section 8.2.4 you learnt about Transmission Mechanisms. Remember that OSEK COM defines two different message transmission mechanisms called `WithCopy` and `WithoutCopy`.

Senders and receivers can, in fact, transfer messages in mixed modes. So, messages could, for example, be sent in `WithoutCopy` mode and received in `WithCopy` mode. This is called **Mixed Mode Transmission**.

8.8 Activating Tasks on Message Transmission

When a message is sent, a task can be activated. Only one task can be activated for each message.

The task that is activated is normally a receiver of the message, but it does not have to be.

If you are going to analyze your application for timing correctness, make sure that you only activate tasks of a lower priority than the message sender. In other words, upward activation of tasks is not allowed.

Figure 8:12 shows that `Task1` has been selected as the task to be activated when `Message1` is sent.

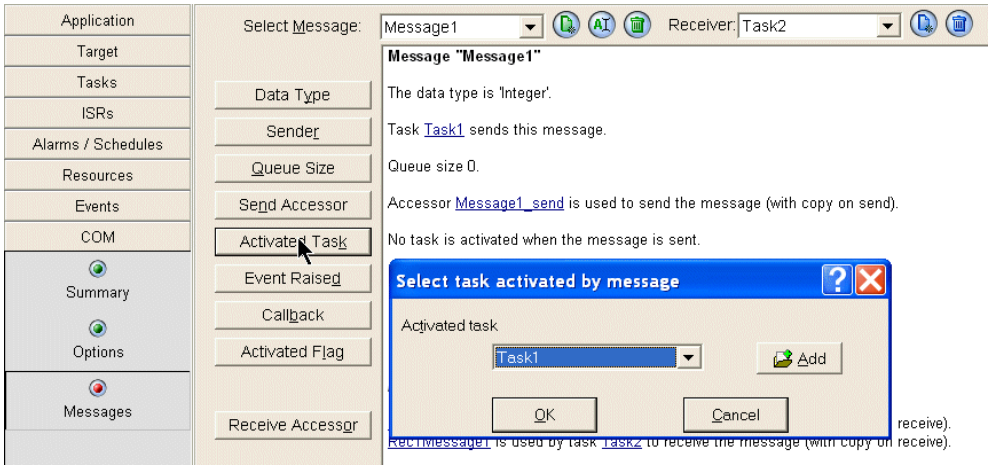


Figure 8:12 - Activating a Task when a Message is sent

8.9 Setting Events on Message Transmission

If a message must be received by an extended task, it is possible to notify the task by setting an event when the message is sent. Figure 8:13 shows how this can be achieved using the RTA-OSEK GUI.

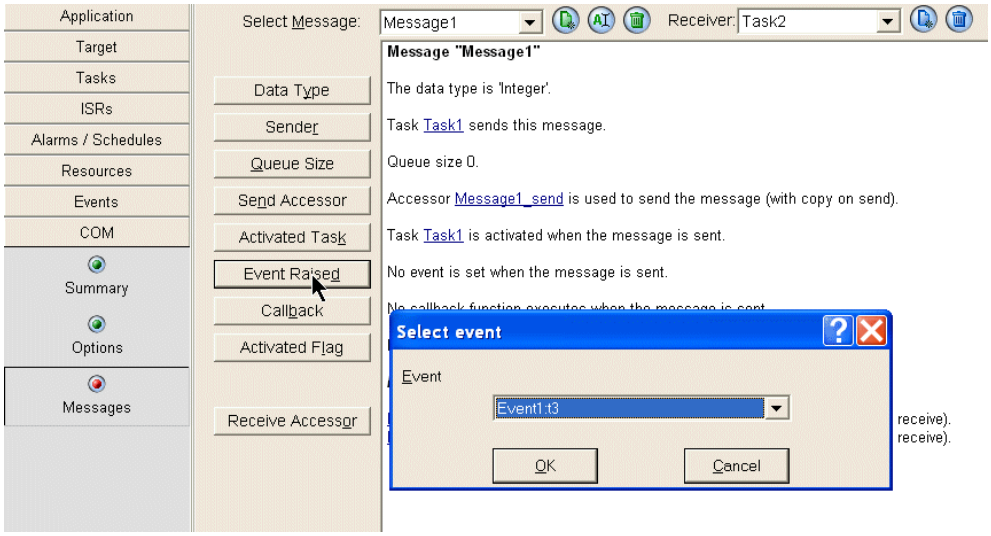


Figure 8:13 - Setting an Event on Message Transmission

8.10 Callback Routines

Messages can have **callback** routines. A callback is a parameterless C function that is called by RTA-OSEK Component when a message is sent. It is up to you to supply this C function.

An example callback routine is shown in Code Example 8:5.

```
void MyCallback (void) {
  /* Callback code. */
}
```

Code Example 8:5 - Writing a Callback Routine

If, for example, you wanted to log a count of message transmissions during debugging, you could create a callback routine to increment a counter on each call.

Have a look at Figure 8:14, where a callback has been specified in the RTA-OSEK GUI.

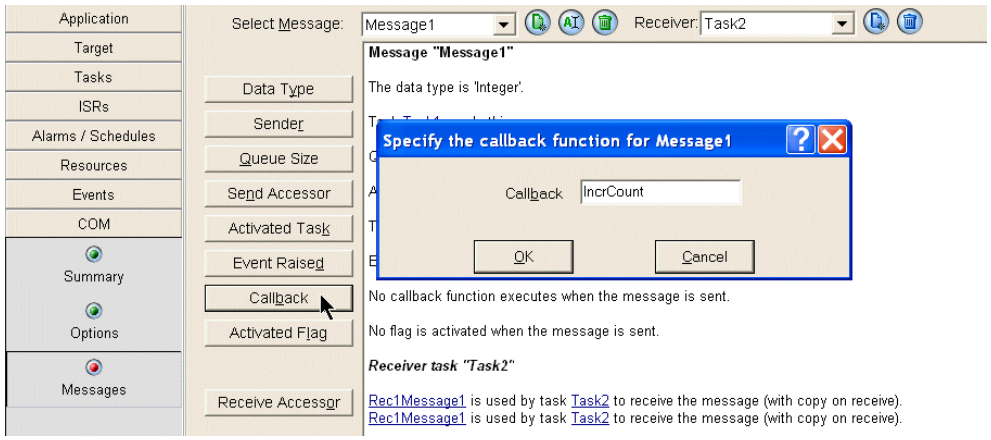


Figure 8:14 - Specifying a Callback Function for a Message

The callback routine can only use the `SuspendAllInterupts()` and `ResumeAllInterrupts()` API calls.

8.11 Using Flags

Flags have a Boolean state, so they can either be set or unset. They are used to manage synchronization with messages. A flag can be set when a message is sent. This flag can be used wherever it is necessary to check for new messages before calling `ReceiveMessage()`.

The flag name, `Flag1`, has been specified in Figure 8:15 for `Message1`.

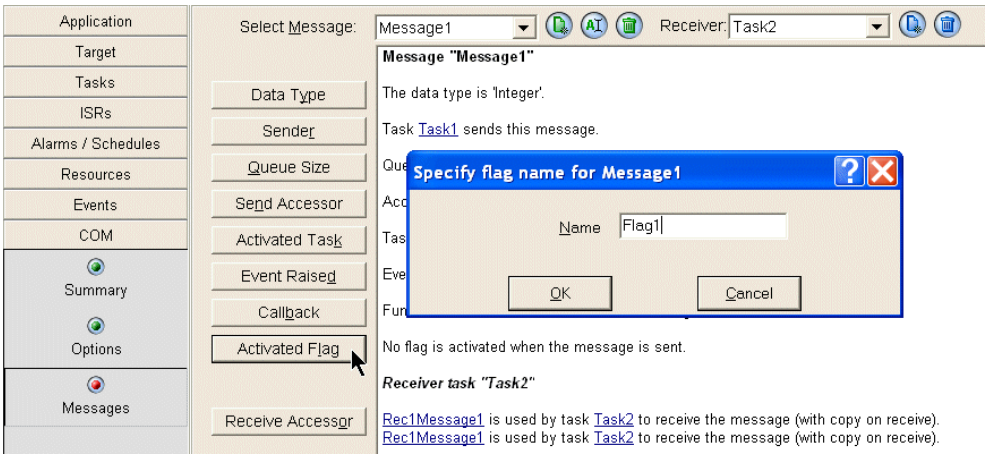


Figure 8:15 - Activating a Flag when a Message is Sent

There are two API calls that allow access to a message flag. `ReadFlag()` returns the current state for a given flag and `ResetFlag()` clears the flag.

Important: It is your responsibility to make sure that the correct flag is being used. There are no checks to ensure that the flag name is correct.

Code Example 8:6 shows the code that is needed for a task to receive a message using an attached flag.

```
TASK(ProcessData)
{
  char buffer [8];

  /* Only receive message if the flag is set. */

  if( ReadFlag( DataHasArrived ) ) {

    /* Receive the message. */
    ReceiveMessage(DataArrived, &ReceiveAccessor);

    /* Retrieve data from accessor. */
    memcpy( buffer, ReceiveAccessor, 8 );
```

```
    /* Reset flag. */
    ResetFlag( DataHasArrived );
  }
}
```

Code Example 8:6 - Task Receiving a Message using the Attached Flag

Messages Summary

- COM provides facilities for message passing between tasks and/or ISRs.
- Non-queued messages have a single sender and multiple receivers. They can be sent `WithCopy` or `WithoutCopy`.
- Queued messages have a single sender and a single receiver. They can only be sent `WithCopy`.
- Message sending and receiving is achieved using accessors.
- You must ensure correct concurrency control when using `WithoutCopy` and queued messages.

9 Introduction to System Modeling

So far you have seen how RTA-OSEK is used in the development process. You have also seen how you can configure and use various operating system objects.

In this chapter you will learn how you can model and build timing relationships into your application.

Real-time systems receive inputs and generate outputs. In RTA-OSEK, the inputs are called **stimuli** and the outputs are called **responses**.

To build a successful real-time system you should be able to answer the following questions:

- Which outputs are related to which inputs?
- How often do inputs occur?

The RTA-OSEK GUI captures this simple information in a stimulus/response model. If you want to analyze a system, however, you will need more information. You will learn more about this later in this guide.

Using the RTA-OSEK GUI you can map your initial specification onto a design, in terms of system objects (tasks, interrupts, alarms, schedules and so on). This design can be analyzed for timing correctness. RTA-OSEK then generates code to implement the design, along with the functional code that is provided by you.

In this chapter you'll see the specification process that should be used when you design systems that use counters, alarms and schedules.

When the development process is complete, each stimulus will be associated with a **primary profile** and each response will be associated with either a primary profile or **activated profile**.

Stimuli can be either external or internal to your system. An external stimulus could be, for example, a press of a button. An internal stimulus could be a timer interrupt from the target hardware.

Usually stimuli originate as interrupts in your application. The interrupt itself may be a stimulus or it could be used by RTA-OSEK Component to generate internal stimuli, such as generating an alarm. During the design process you will decide what form the stimulus takes.

When a stimulus occurs, one or more responses must be generated in the system. As with stimuli, responses can be external or internal to your system. An external response could be the actuation of some hardware. An internal response may be the availability of some calculation. During the design you will decide what your program will do to generate the responses.

9.1 Declaring Stimuli and Responses

The first part of any specification involves declaring the stimuli in the system, along with their associated responses.

Each stimulus and response must have a unique name. When you declare a stimulus, the RTA-OSEK GUI generates a default response with the same name as the stimulus. Usually you will want to change this name.

Figure 9:1 shows how the response name is changed.

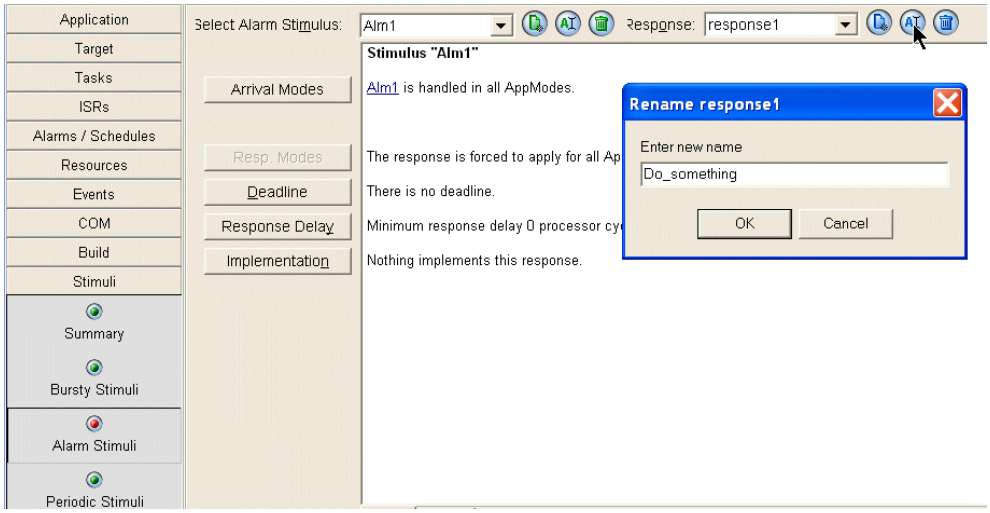


Figure 9:1 - Renaming the Default Response

A stimulus can be associated with multiple responses. Each response must have a unique name. For example, a stimulus called `10ms_stimulus` may have an associated response called `checked_vessel_pressure`. A stimulus called `brake_pressed` may have responses called `hydraulics_primed`, `pads_applied` and `brake_lights_on`.

9.2 Arrival Patterns and Arrival Rates

Declaring the stimuli and responses in your system specifies which inputs are related to which outputs. Once the stimuli and responses have been declared, the next thing to do is to specify how often the stimuli occur.

Each stimulus has an **arrival pattern**. The arrival patterns can be:

- Bursting.
  A bursting arrival pattern is used to model the case where, generally, an interrupt is a stimulus and it is used to directly activate a task to generate a response.
- Periodic.
  A periodic arrival pattern is used to model a periodic rate, for example when you want to generate a response every 20ms.
- Planned (aperiodic).

  A planned arrival pattern is used to model an aperiodic series of stimuli. For example, you may want to generate a response at 10ms, 15ms, 50ms and so on.

In the case of both periodic and planned arrival patterns, it is the behavior that is being specified. During the design process it is up to you to decide how to achieve the specified behavior at run-time.

An example of each type of arrival pattern is shown in Figure 9:2.

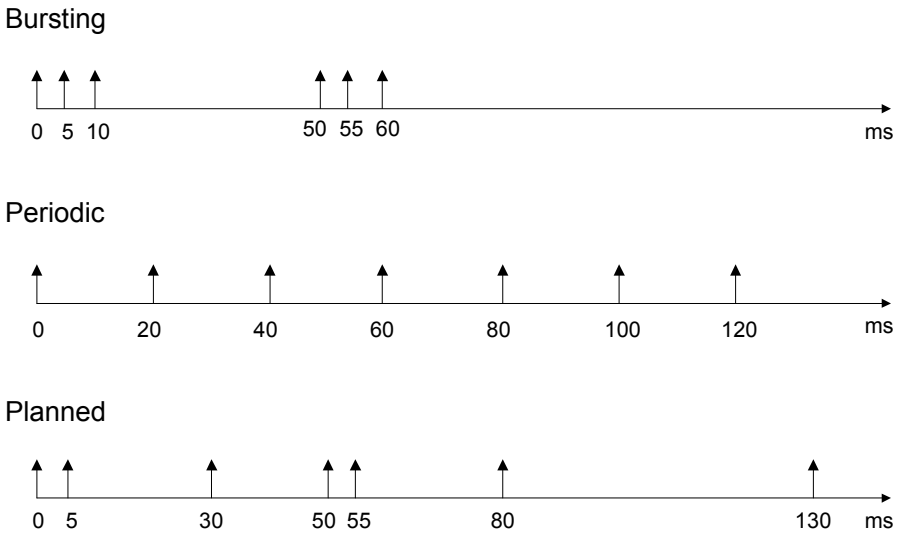


Figure 9:2 - Stimuli Arrival Pattern Examples

Each arrival pattern has an associated **arrival rate** that specifies the required timing behavior of the arrival pattern.

Bursting patterns are used only for timing analysis. Periodic arrival patterns are used for analysis, as well as for generation of run-time data used by your application. For planned arrivals, the actual timing specification is deferred until the design stage. The plan that you create states when stimuli occur.

9.3 Designing Responses

Each response that you declared during specification must be associated with an implementation. The implementation of a response is performed in functional code that you provide.

Responses can be generated by any functional code in your application. You will also need to declare which task or ISR will implement the response. A task or ISR can implement more than one response, an example is shown in Figure 9:3.

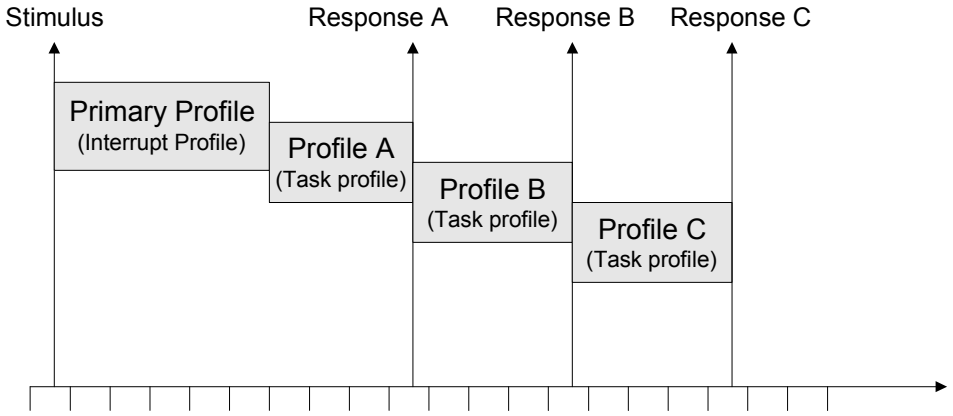


Figure 9:3 - Implementation of a Stimulus and Responses

Note that an ISR can both react to a stimulus and make a response. For example an interrupt handler may service the interrupt source, perform some processing and then return a result to the real-world.

9.4 Designing Stimuli

Bursting stimuli are usually generated by ISRs, specified as the primary profile. This is the only thing that needs to be specified when you create a bursting stimulus.

For periodic and planned stimuli you need to decide how the stimulus is going to be generated in RTA-OSEK Component.

Figure 9:4 shows you how you can visualize the design of stimuli.

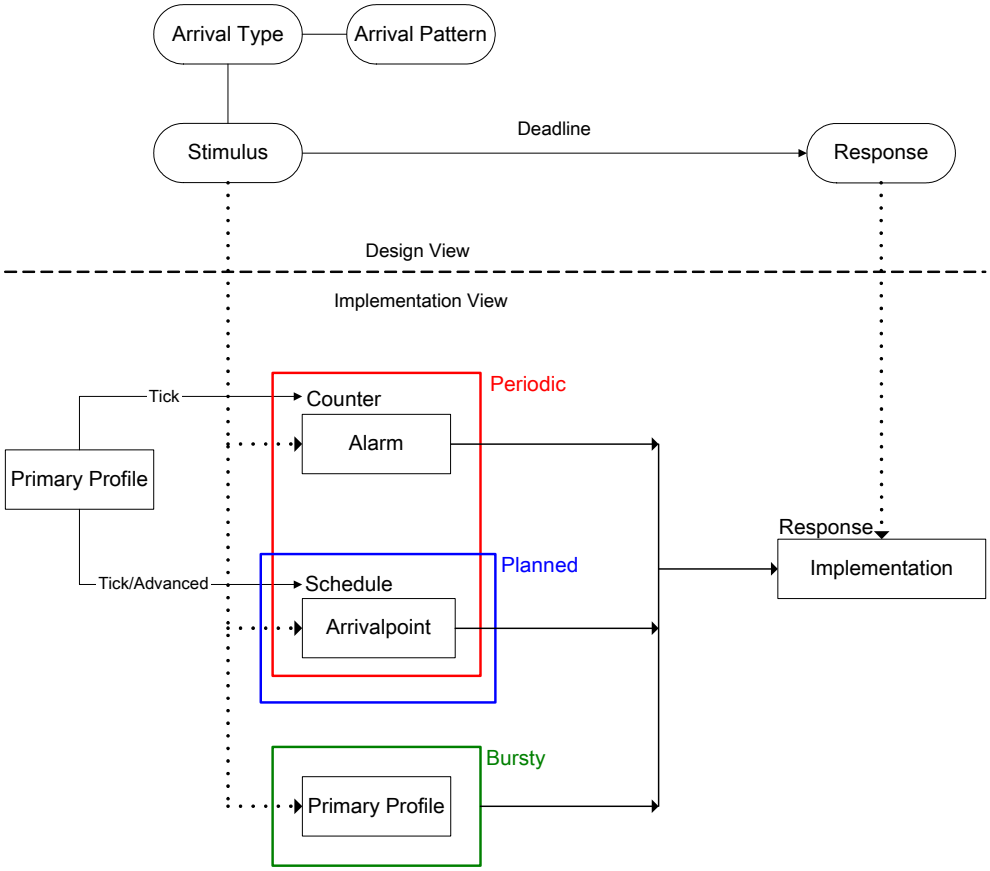


Figure 9:4 - Designing Stimuli

Introduction to System Modeling Summary

- All practical systems have inputs and outputs. In RTA-OSEK, the inputs are called stimuli and the outputs are called responses.
- A stimulus is associated with at least one response.
- Responses are implemented by tasks or ISRs in your final application code.

- Stimuli have arrival types and arrival patterns. Arrival information is used for analysis and, in the case of periodic and planned arrivals, for the generation of run-time information.
- For periodic and planned stimuli you must design how the stimuli will arrive in your application when running under RTA-OSEK Component.

10 Counters and Alarms

It is possible to construct systems that activate tasks at different rates using ISRs. However, for complex systems, this can become inefficient and impractical.

OSEK provides a systematic way for constructing larger and more complex systems using the counter/alarm mechanism.

10.1 The Counter/Alarm Mechanism

The counter/alarm mechanism consists of two parts:

- A counter.
  The counter registers a number of **ticks**.
  This could be, for instance, the number of 1ms interrupts that the system has received or the number of times that a switch has been pressed.
  Note that a tick is not necessarily a timer tick. In an event-based operating system, such as OSEK, the tick can be anything that you can capture in the system.
- A number of alarms attached to the counter.
  The alarm part specifies an action (or actions) to perform when a particular counter value is reached. Each counter in your system can have any number of alarms attached to it.

An ISR (or sometimes a task) is used to drive a counter. The driver is responsible for making the correct RTA-OSEK Component API call to 'tick' the counter.

An alarm is said to have **expired** when the value of a counter equals the value of an alarm attached to the counter. On expiry RTA-OSEK Component will perform the action associated with the alarm. The action could be to activate a task, to execute an alarm callback routine or to set an event.

Portability: In OSEK each alarm can activate a task, set an event or execute a callback function. In RTA-OSEK, however, you have more flexibility. You can activate a task, set an event and execute a callback function from a single alarm.

The alarm expiry value can be defined relative to the actual counter value or as an absolute value. If the alarm expiry is defined as relative to the actual counter, it is known as a **relative alarm**. If it is defined as an absolute value, it is known as an **absolute alarm**.

Alarms can be configured to expire once. An alarm that expires once is called a **single-shot alarm**.

An alarm can also be specified to expire on a periodic basis. This type of alarm is called a **cyclic alarm**. You can find out more about cyclic alarms in Section 10.5.

10.2 Declaring Counters

Counters are declared using the RTA-OSEK GUI. To declare a counter you must specify:

- A counter name.
  RTA-OSEK creates a handle for each counter using an identifier of the same name as the counter.
- The rate at which the counter is ticked.
- A primary profile.
  This is usually the interrupt that you are using to tick the counter.

Figure 10:1 shows how a counter called `Counter1` has been declared.

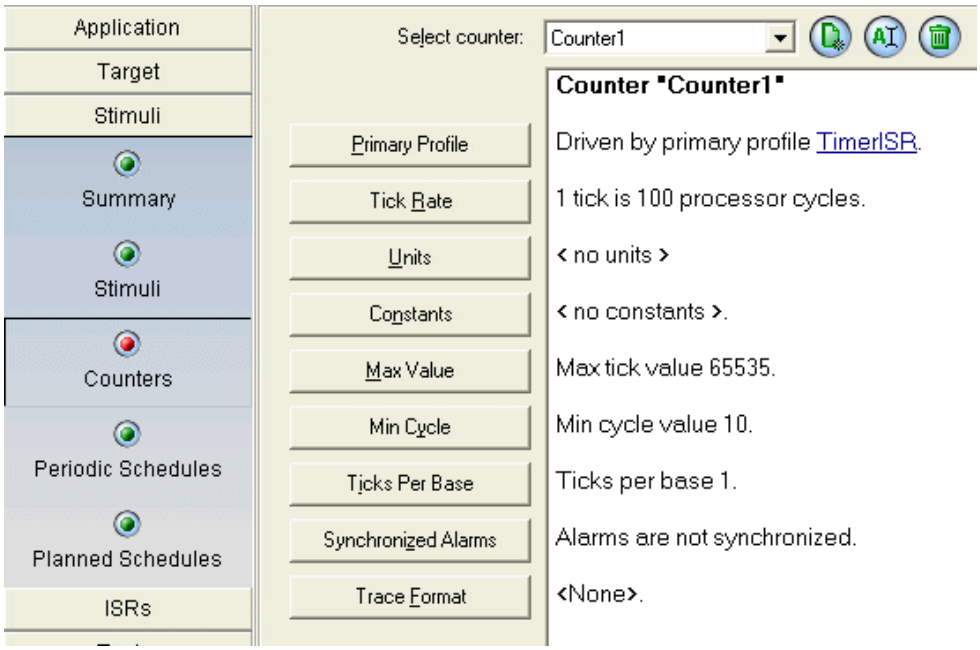


Figure 10:1 - Declaring a Counter

10.3 Declaring Alarms

In RTArchitect an alarm is not declared directly. Alarms are created by:

- Declaring a stimulus.
- Attaching the stimulus to a counter.

When a stimulus is attached to a counter it becomes an alarm on the counter. The response to the stimulus becomes the action performed when the alarm expires. It is also possible to specify that an event and a callback routine are attached to the alarm.

The alarm action(s) can be to raise an event, execute a callback routine and/or activate a task on expiry. If you need to set multiple events, to make multiple callbacks or to activate multiple tasks on expiry, you will need multiple alarms with the same expiry value. (SSX5 schedules provide an alternative mechanism for implementing multiple task activation without the need for multiple alarm objects. You will learn about schedules later in this guide).

Important: Only periodic stimuli can be attached to a counter. You are not, however, limited to periodic alarms in the implementation. Alarm periods can be set to any value at run-time*.

In Figure 10:2, `Stimulus1` is created implemented as an Alarm attached to `Counter1`.

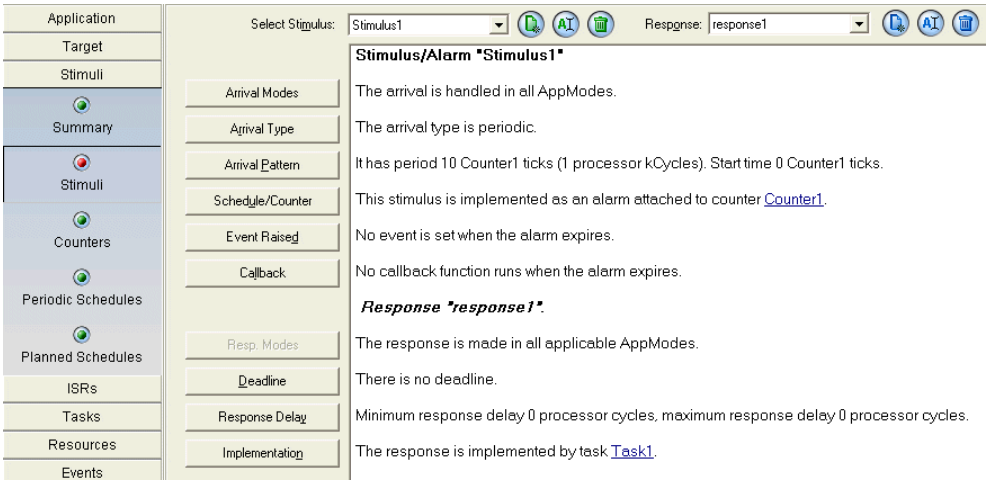


Figure 10:2 - Creating an Alarm

The RTA-OSEK GUI can be used to show a graphical display of each counter. You can see this by selecting the **Graphic** tab in the **Counters** workspace. The visualization shows the alarms that are attached to each counter. An example of the graphical view is shown in Figure 10:3.

* If you need to perform timing analysis on aperiodic alarms, then you will have to specify the shortest period in the alarm declaration.

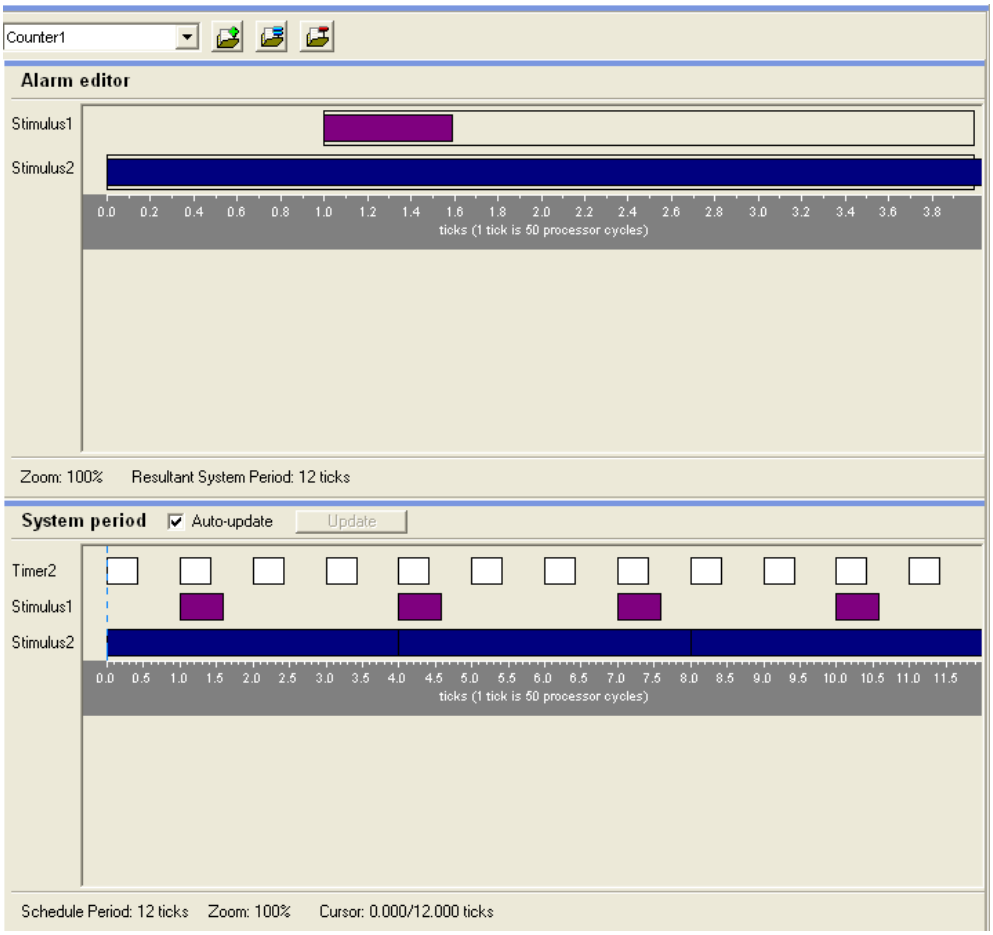

Counter1
Alarm editor
Stimulus1
Stimulus2
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8
ticks (1 tick is 50 processor cycles)
Zoom: 100%
Resultant System Period: 12 ticks
System period
Auto-update
Update
Timer2
Stimulus1
Stimulus2
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0 10.5 11.0 11.5
ticks (1 tick is 50 processor cycles)
Schedule Period: 12 ticks
Zoom: 100%
Cursor: 0.000/12.000 ticks

Figure 10:3 - Viewing the Counter Graphically

In the graphical view you can manipulate alarms on the counter by dragging the alarms to new locations.

10.3.1 Alarm Actions

Each alarm that you create is associated with up to 3 actions:

- Activate a task.
- Raise an event.
- Execute a callback function.

When you attach a stimulus to a counter the response becomes the default action. This action is usually to activate a task. If the alarm raises an event or executes a callback then, when the alarm expires, these will also occur.

To create an alarm that only has an action of setting an event or raising an alarm, you should create a stimulus/response model with no specified implementation. You should then attach the stimulus to your counter, specify that an event is raised or the callback is executed and then set the implementation to be the event being raised and/or callback executed.

If the stimulus attached to an alarm has multiple responses, which are implemented by multiple tasks, then only the highest priority task will be attached to the alarm action.

It is up to you to make sure that the other responses are generated by using chained activations of tasks. The RTA-OSEK GUI tells you that you must use direct activation chains to implement this activation scheme.

10.3.2 Autostarting Alarms

Alarms can be autostarted during `StartOS()`. When you create an alarm it is automatically assigned a start time of 0 ticks. The start time specifies the time between the counter being started and the first expiry of the alarm. The start time is used at run-time, but only when alarms are autostarted.

If you specify that your alarms are synchronized and you intend to perform timing analysis on your application, you must make sure that the start time is less than the period of the alarm.

Alarms are set automatically using absolute values. At `StartOS()` the underlying counter is set to have an initial count value of 0 ticks. As a result of this, you must take care if you use the default start time of 0 ticks. The 0th tick has already happened when the alarms are started, so the first expiry of an alarm will not occur until the associated counter has wrapped around.

Autostarted alarms should be used if you specify that alarms are synchronized. During `StartOS()`, RTA-OSEK Component will make sure that all autostarted alarms for a counter are synchronized at startup.

10.4 Incrementing Counters

When an alarm is used in an RTA-OSEK application you must increment the corresponding counter.

RTA-OSEK generates a `Tick_<CounterID>()` API call for each counter that has been declared in the configuration file (where `CounterID` is the name of the counter).

Portability: The OSEK OS standard does not specify a standard API call for dealing with counters. If you are porting your application from another OSEK to RTA-OSEK, the counter tick API calls must be changed.

The `Tick_<CounterID>()` API call must be made whenever you have to increment the counter. Although there is no restriction on where and when a counter can be incremented, it is usually implemented in a Category 2 ISR handler. A task could, however, also make this API call.

Let's look at an example. An application contains two counters, one called `TimeCounter` and one called `AngularCounter`. RTA-OSEK will generate the two API calls, shown in Code Example 10:1.

```
Tick_TimeCounter();
Tick_AngularCounter();
```

Code Example 10:1 - Sample Counter API Calls Generated by RTA-OSEK

The interrupt handlers that you supply to service the timer and angular interrupts must call these API calls.

Code Example 10:2 shows how these interrupt handlers could look.

```
#include HandleTimerInterrupt.h

ISR(HandleTimerInterrupt) {

  ServiceTimerInterrupt();
  Tick_TimeCounter();

}
```

```
#include HandleAngularInterrupt.h

ISR(HandleAngularInterrupt) {

  ServiceAngularInterrupt();
  Tick_AngularCounter();

}
```

Code Example 10:2 - Interrupt Handlers for Code Example 10:1

10.5 Setting Alarms

Two API calls are provided for setting alarms:

- `SetRelAlarm(AlarmID, Increment, Cycle);`
  Sets the alarm to expire `Increment` number of ticks relative to the time at which you make the call. This means that, `Increment` is a tick offset from the current counter tick value.
- `SetAbsAlarm(AlarmID, Start, Cycle);`
  Sets the alarm to expire when the counter value next reaches the value `Start`. You should be aware that if the counter has just ticked by this value, it has to 'wrap around'. This means that when it reaches its maximum value it will have to count up again from 0 until the expiry is reached.

In these two API calls, a `Cycle` value of zero ticks indicates that the alarm is a single-shot alarm, which means that it will expire only once before being cancelled. A `Cycle` value greater than zero defines a cyclic alarm. This means that it will continue expiring, at the rate specified, after the first expiry has occurred.

The RTA-OSEK GUI gives implementation guidelines that specify the cycle rates for alarms. An example of the implementation guidelines is shown in Figure 10:4.

```
Stimuli
Task "Task1" will be activated when alarm "Stimulus1" expires.
 Task "Task1" must directly activate task "Task2".
 Alarm "Stimulus1" must be started by 'SetRelAlarm(Stimulus1,0,5)' or 'SetAbsAlarm(Stimulus1,0,5)'.

Counters
ISR "CounterTick" drives counter "Counter1".
 It must call 'Tick_Counter1()' every 1 real time ms.
 It drives alarm "Stimulus1".

Cat2 ISRs
ISR "CounterTick" must run at priority 1 and respond to interrupts on vector 'Real time interrupt'.
"CounterTick" must service a single interrupt source and then exit.
"CounterTick" must tick counter "Counter1" every 1 real time ms.
 e.g.
        #include "CounterTick.h"
        ISR(CounterTick)
        {
                service_interrupt();
                Tick_Counter1();

        }

Tasks
```

Figure 10:4 - Implementation Guidelines

10.5.1 Alarm Examples

Have a look at the following code examples and the timelines used to illustrate the alarms.

A Relative Cyclic Alarm

Code Example 10:3 shows a relative alarm that expires after 30 ticks and then every 75 ticks thereafter.

```
/* Expire after 30 ticks, then every 75 ticks. */
SetRelAlarm (Alarm1, 30, 75);
```

Code Example 10:3 - Relative Cyclic Alarm (with 75 ticks cycle)

In Figure 10:5, you can see how this alarm can be visualized.

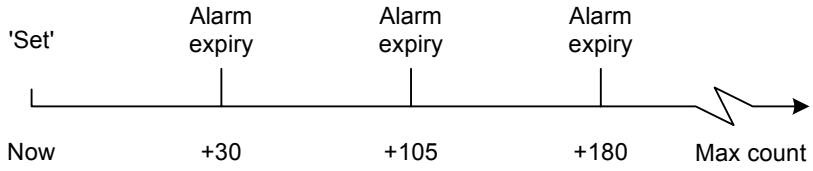


Figure 10:5 - Illustration of the Alarm in Code Example 10:3

A Single Relative Alarm

When setting a relative alarm with `Increment` value equal to zero, the alarm will expire after the current counter value is reached again, in other words, after a full revolution of the counter.

Code Example 10:4 shows how a single relative alarm is set.

```
/* Expire after Max Value ticks. */
SetRelAlarm (Alarm2, 0, 0);
```

Code Example 10:4 - Setting a Relative Single Alarm

This can be shown, simply, in Figure 10:6.

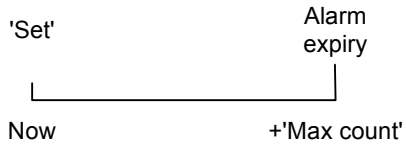


Figure 10:6 - Illustration of the Alarm in Code Example 10:4

An Absolute Alarm

For absolute alarms, an absolute start value of zero ticks is treated in the same way as any other value.

If, for example, the current alarm counter value was zero it would need the maximum counter value ticks until the next expiry. If it were already at the maximum value, however, it would only take one tick.

Code Example 10:5 shows how an absolute cyclic alarm can be set.

```
/* Expire when counter value reaches 42. */
SetAbsAlarm (Alarm3, 42, 0);
```

Code Example 10:5 - Example Absolute Cyclic Alarm

Code Example 10:5 is illustrated in Figure 10:7.



Figure 10:7 - Illustration of the Alarm in Code Example 10:5

Important: Specifying a very small relative increment or an absolute start value that is very close to the current counter value may cause undesired side effects. The alarm could go off while the task is still executing.

If the activated task is BCC1 or ECC1 there will be no queued activation and several task executions could potentially be 'lost'. You must make sure that enough time is allowed for the task or ISR handler to complete before the first alarm is due to expire.

Often the best way to validate the current alarm counter is to use the `GetAlarm()` API call, which returns the relative number of ticks until the next expiry (in Section 10.13 you'll find out how to determine the next alarm expiry). If the time appears too long, it is then possible to cancel and reset the alarm (see Section 10.6 to find out how to cancel alarms).

10.6 Canceling Alarms

You can cancel an alarm using the `CancelAlarm()` API call.

An alarm may, for example, need to be cancelled to stop a particular task being executed. An alarm can be restarted using the `SetAbsAlarm()` or the `SetRelAlarm()` API call.

Advanced Counter and Alarm Concepts

10.7 Alarm Callbacks

Each alarm can have an associated callback function. The callback is simply a C function that is called when the alarm expires.

Each callback routine must be written using the `ALARMCALLBACK()` macro, shown in Code Example 10:6.

```
ALARMCALLBACK(MyCallback) {
  /* Callback code. */
}
```

Code Example 10:6 - Writing a Callback Routine

Callback routines run at OS level, which means Category 2 interrupts are disabled. The only RTA-OSEK Component API calls that you can make are the `SuspendAllInterrupts()` and `ResumeAllInterrupts()` calls.

Figure 10:8 shows how to configure a callback routine for an alarm.

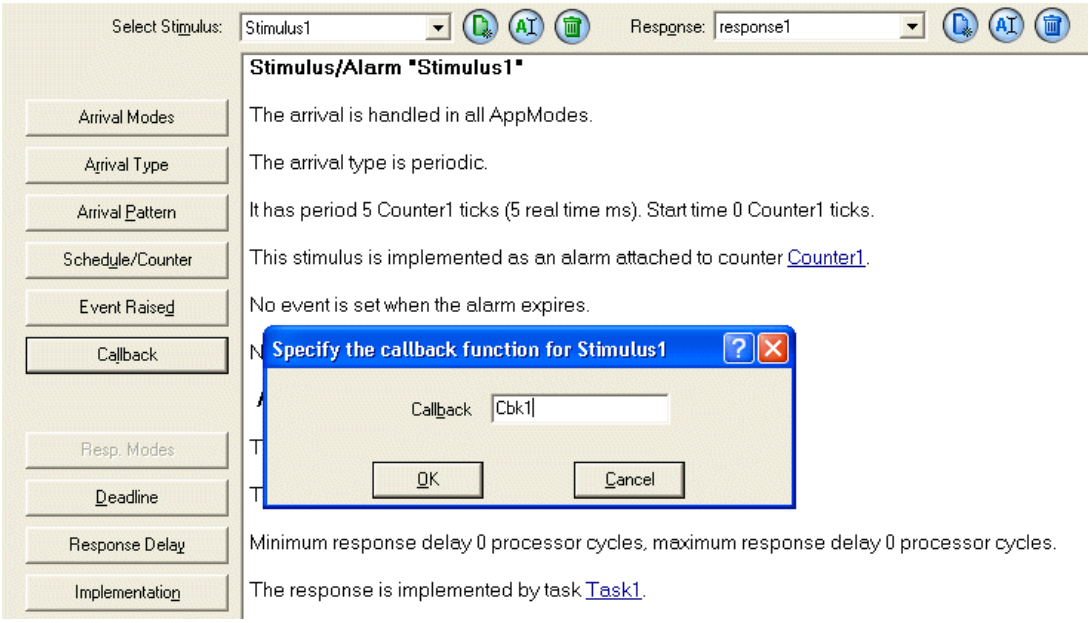


Figure 10:8 - Configuring a Callback Routine for an Alarm

10.8 Setting an Event

Each alarm can set an event for a specified task. When an event is set with an alarm, it has the same properties as it would if it were set using the `SetEvent()` API call. Figure 10:9 shows you how to set an event action for an alarm.

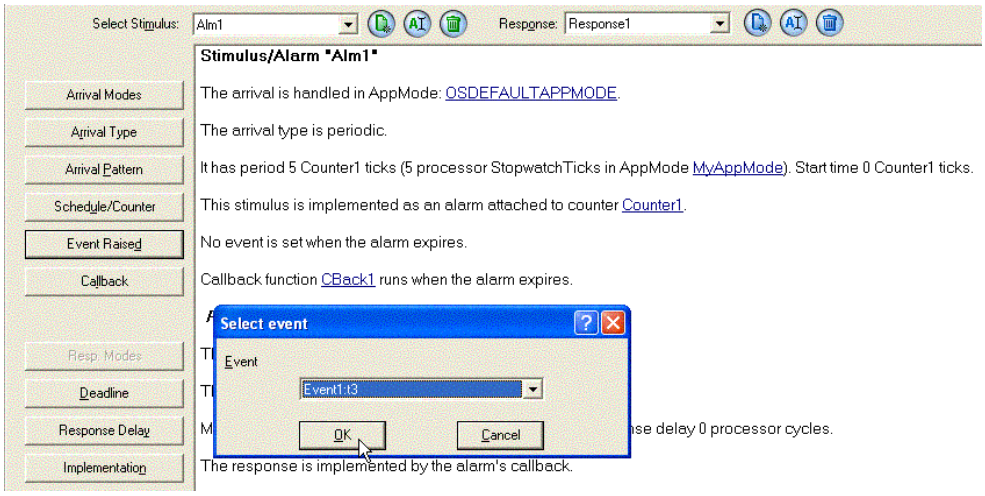


Figure 10:9 - Setting an Event Action for an Alarm

10.9 Releasing Multiple Tasks from Alarms

In OSEK you may only activate a single task for each alarm. If you have multiple responses to a stimulus and the stimulus is implemented using a counter, only the highest priority response implementation profile will be activated when the alarm expires.

To achieve the required behavior, the code for the highest priority task profile must directly activate the lower priority task profile(s). The implementation notes in the RTA-OSEK GUI will tell you how to do this.

There are a number of ways that you can directly activate the lower priority profiles. You can find out more about these methods later.

10.10 Changing Counter Attributes

Each counter has the following attributes:

- A maximum value.
  Defines the maximum count value for the counter. The default setting is
  target dependent. This corresponds to the OSEK attribute `MAXALLOWEDVALUE`. See the *RTA-OSEK Binding Manual* for your target for further information.
- A minimum cycle value.
  Defines the shortest time unit you can use when setting a cycle value.

By default this is 1 tick. This corresponds to the OSEK attribute `MINCYCLE`.

- Ticks per base.
  You can assign any value to this attribute because it is not used by RTA-OSEK. This corresponds to the OSEK attribute `TICKSPERBASE`.

All of these values can be changed if required. You might, for example, want an 8-bit counter rather than a 16-bit counter. You may also, for instance, want to specify a minimum cycle value to use when debugging. This can prevent the counter being set to a value that has been reached when the set call is made.

The RTA-OSEK Component API call `GetAlarmBase()` always returns the configured counter values. The structure of `GetAlarmBase()` is shown in Code Example 10:7.

```
AlarmBaseType Info;
GetAlarmBase( Alarm2, &Info );

MaxValue  = Info.maxallowedvalue;
BaseTicks = Info.ticksperbase;
MinCycle  = Info.mincycle;
```

Code Example 10:7 - The Return Structure of GetAlarmBase()

The configured values are also available as symbolic constants in the form:

```
OSMAXALLOWEDVALUE_<CounterID>
OSTICKSPERBASE_<CounterID>
OSMINCYCLE_<CounterID>
```

Code Example 10:8 - Symbolic Constants

10.11 Using Non-Time Based Counter Units

So far you have learnt about counters that use time as the tick. In fact, counters can be ticked by any tick source. All alarms attached to the counter will be related to that tick source. Remember that you saw earlier that in an event-based operating system, such as RTA-OSEK Component, the tick could be anything that you can capture in the system.

The RTA-OSEK GUI allows you to declare counter **units**. Units allow you to specify non-time related tick sources in terms of real-time units. The time conversion between the unit and time must represent the worst-case conversion. For example, this could be the fastest rate a button is pressed or the fastest rotation speed of a timing wheel.

Important: If the worst-case conversion rate is incorrectly specified, any analysis you perform on your application will not be accurate.

You could have a counter, for instance, that counts teeth on a toothed timing wheel and then activates tasks at specific angular rotations. In the RTA-OSEK GUI you could declare a degree unit and then specify that there are 360

degrees in a revolution. You can see how this can be achieved in Figure 10:10.

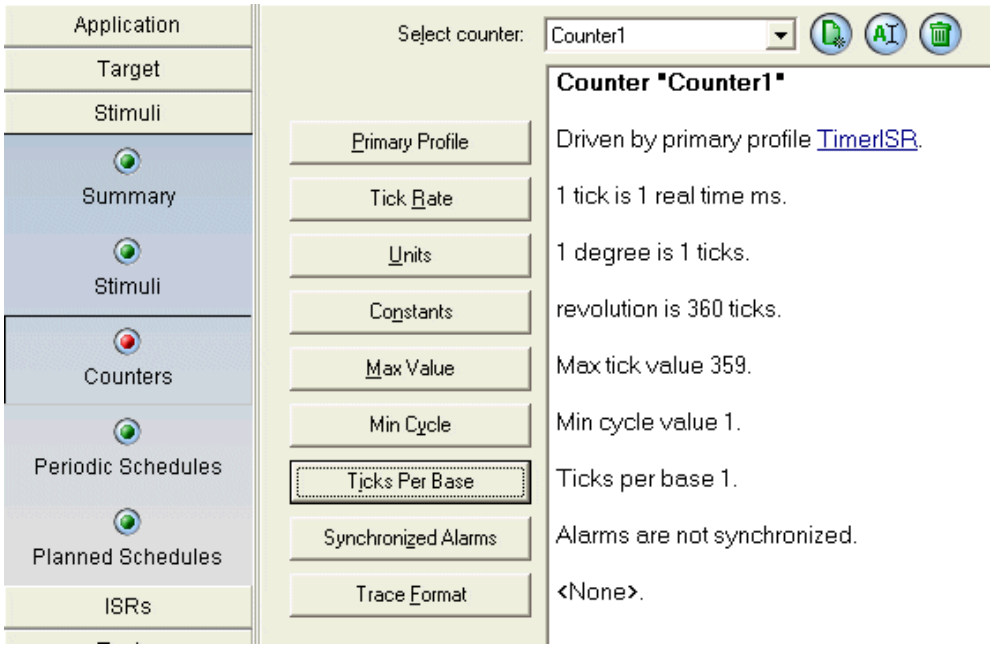


Figure 10:10 - Declaring an Angular Unit

Following on from our example, if you stated that the angular counter uses 5 ticks per degree, for instance, you must make sure that the interrupt source provides 5 ticks for each degree of rotation.

10.12 Specifying Counter Constants

When alarms are being set, it is up to you to specify the start times, increments and cycle times in counter ticks.

The RTA-OSEK GUI allows you to declare symbolic constants for commonly used counter values. Figure 10:11 shows you how a constant has been defined.

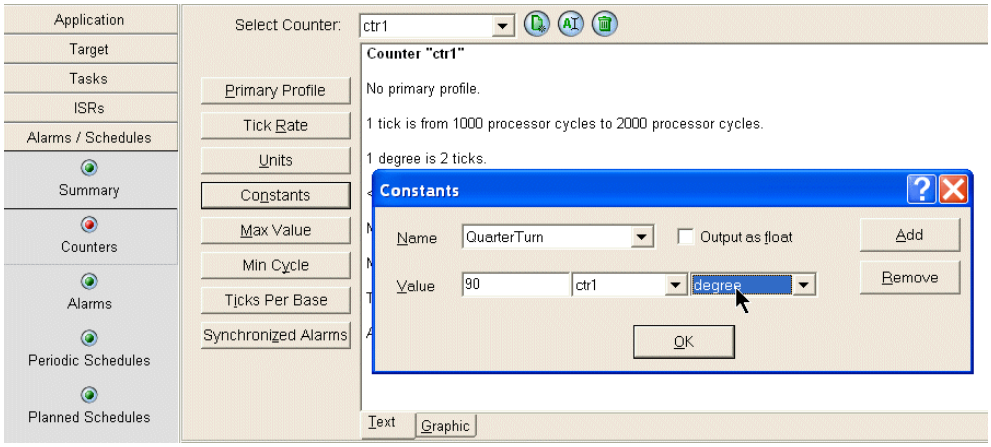


Figure 10:11 - Declaring a Counter Constant

Declared counter constants are available to your application code through the generated header files.

10.13 Determining the Next Alarm Expiry

The RTA-OSEK GUI allows you to determine the amount of time remaining before a particular alarm expires. You can do this, for example, to avoid setting an absolute alarm when the absolute value has already been reached.

The `GetAlarm()` API call allows you to get the number of ticks before the specified alarm expires.

10.14 Autostarting Alarms

It is possible to start cyclic alarms by calling `SetRelAlarm()` or `SetAbsAlarm()` in the main program. However, the easiest way to set cyclic alarms is to make them autostarted. Autostarted alarms will be started during `StartOS()`.

Autostarted alarms can be set on a per application mode basis. Figure 10:12 shows how alarms can be set to autostart from the Startup Modes pane.

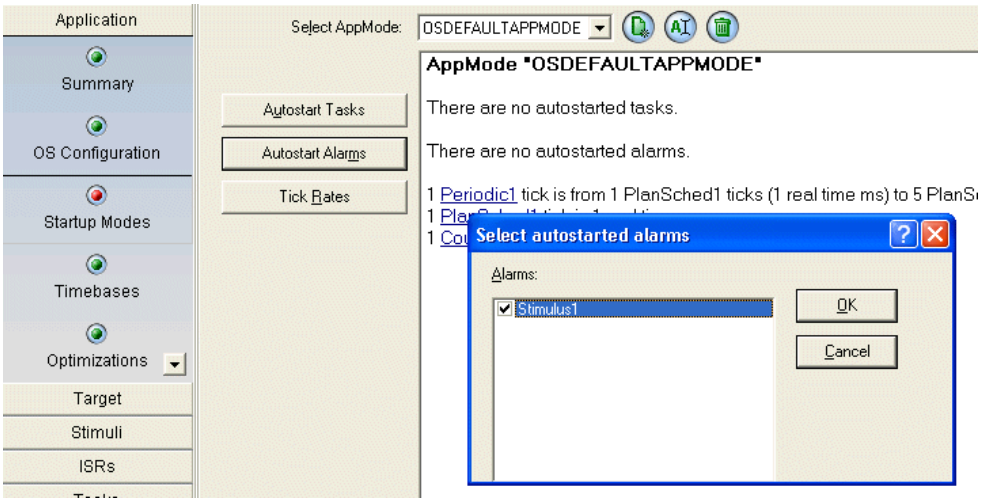


Figure 10:12 - Autostarting Alarms

10.15 Real-time and CPU-time

When you specify the counter tick rate in the RTA-OSEK GUI, you can either specify the ticks in terms of their CPU clock rate or in terms of real-time (nanoseconds, microseconds and milliseconds for example).

For most of the applications that you write, the relative timing of events will be the real-time values determined by your system requirements. This means that you will usually specify alarm and counter values in terms of real-time units.

Using these units has an important advantage. If you use real-time units and then change the CPU clock rate, the counter timing values will be scaled automatically according to the new CPU clock rate.

10.16 Synchronization using Alarms

The safest and easiest way to synchronize alarms is to set a number of absolute alarms that are linked to the same counter. Using absolute alarms does not affect the start time. This avoids potential problems with interfering interrupts.

Synchronizing relative alarms is more complex because intermittent delays, due to interrupts and preemption, can result in different alarm offsets being set on startup.

Alarm synchronization can be selected in the RTA-OSEK GUI during the configuration of a counter. This is shown in Figure 10:13.

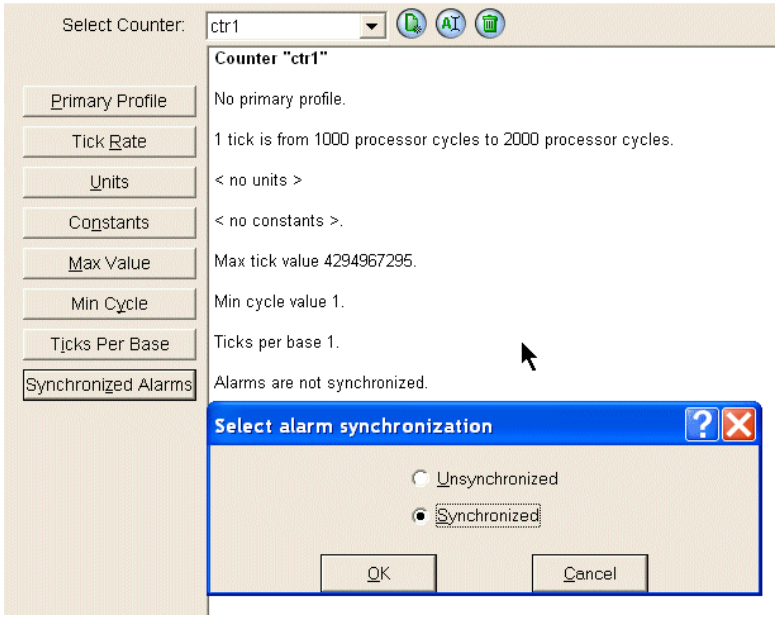


Figure 10:13 - Alarm Synchronization

To ensure synchronization between alarms on a counter at run-time, you must make sure that the alarms are autostarted. If an alarm is cancelled it must also be reset with a `SetAbsAlarm()` call.

Important: If alarms are synchronized in the RTA-OSEK GUI this does not guarantee that the alarms are actually synchronized. It simply informs the RTA-OSEK Planner that you will guarantee synchronization.

If you intend to build a system for timing analysis, it is better if you can guarantee synchronization. If synchronization is important, consider using RTA-OSEK schedules. Schedules guarantee synchronization and offer a flexible approach to the design of event-based hard real-time systems. You can find out more about schedules later in this guide.

10.17 Absolute Periodic Alarms

In OSEK, it is not possible to set an absolute periodic alarm relative to the maximum counter value.

For example, if a counter has a modulus of 65536 ticks, you cannot use the `SetAbsAlarm(Alarm1, 10, 65536)` call to set an absolute periodic alarm that expires 10 ticks after the counter has wrapped around. This is forbidden by the OSEK standard because the cycle value cannot be greater than `OSMAXALLOWEDVALUE`, which is a maximum of 65535.

If you need this functionality, you must provide code that resets an absolute single-shot alarm each time the alarm expires.

For example, if `Task1` is attached to `Alarm1`, then the body of `Task1` will need reset the alarm when the task is activated. Have a look at Code Example 10:9.

```
TASK(Task1) {

  /* Single-shot alarm reset. */
  SetAbsAlarm(Alarm1, 10, 0);

  /* User code. */
  TerminateTask();

}
```

Code Example 10:9 - Resetting an Alarm when a Task is Activated

10.18 Aperiodic Alarms

The RTA-OSEK GUI will suggest the implementation that you should use when you create a series of alarms. The suggestions will show you what you should do to give the specified timing behavior.

To achieve aperiodic behavior you should use single-shot alarms that are set to the next expiry value by the activated task.

In Code Example 10:10, the alarm must expire after 10 ticks, then after a further 12 ticks. The alarm activates task `Task1`. The first alarm is autostarted by RTA-OSEK Component. In `Task1` the alarm has to be reset for the next expiry.

```
TASK(Task1) {

  SetRelAlarm(Alarm1, 10, 0);
  /* Rest of task. */

}
```

Code Example 10:10 - Aperiodic Alarm Example

If you choose to use this method then you must ensure that the times specified in the stimulus/response model represent the shortest time between successive alarm expiries.

Counters and Alarms Summary

- Counters are used in OSEK to register a count of some tick source.
- Counters can count any tick value and RTA-OSEK allows you to specify the actual counter units.
- Each counter can have a number of alarms that specify when some action (activation of a task, setting an event, execution of a callback) must be carried out.
- Alarms can be autostarted.
- Alarms can be synchronized.

11 Schedules

You saw earlier that the OSEK standard provides alarms and counters. They can be used to construct systems that require recurring task activations. In addition to this, however, RTA-OSEK also provides schedules. Schedules provide more flexibility than the counter/alarm mechanism.

11.1 Using Schedules

If you use schedules, you can configure systems where multiple tasks can be released at a single point in time (rather than having to specify multiple alarms). You can also change the relative times between task activations, whilst retaining synchronization across the whole schedule.

Portability: Schedules are provided by RTA-OSEK for building and controlling complex systems. Schedules are not part of the OSEK OS standard.

11.1.1 Types of Schedules

There are two types of schedule:

- Periodic.
  A periodic schedule allows you to implement periodic stimuli.
- Planned.
  A planned schedule allows you to implement aperiodic stimuli.

Planned schedules provide much more flexibility than periodic schedules. You can switch in and switch out of sections of the schedule at run-time and specify that the whole schedule is single-shot (to allow for the phased release of a number of sequences of tasks, for example).

11.1.2 Arrivalpoints

Periodic and planned schedules consist of a series of **arrivalpoints** and a set of state variables. When a schedule reaches an arrivalpoint it is said to have **arrived**.

The arrivalpoints for a periodic schedule are implicit. They are automatically generated by RTA-OSEK and used internally by RTA-OSEK Component. For planned schedules, however, you must configure the arrivalpoints yourself.

Arrivalpoints are similar to alarms. They are used to implement stimuli in the system; however, arrivalpoints differ from alarms in a number of ways:

- An arrivalpoint can implement multiple stimuli (i.e. dispatch multiple tasks).
- When a stimulus generates multiple responses, RTA-OSEK Component manages the activation of multiple tasks to generate the responses. This means that you don't need to implement the chained task activation that is required when using the counter/alarm mechanism.

- Arrivalpoints cannot set events or make callbacks.

Arrivalpoints have the following properties:

- A set of stimuli to trigger on arrival.
- A delay until the next arrivalpoint occurs.
- A next arrivalpoint.
- A set of analysis attributes.

For each stimulus associated with an arrivalpoint, the responses triggered by the stimulus will be released on arrival. The responses to the stimulus must be generated by tasks. At run-time, RTA-OSEK Component will simultaneously release all the tasks associated with all the stimuli on an arrivalpoint.

When a stimulus is attached to an arrivalpoint, the arrivalpoint becomes the implementation of the stimulus at run-time. The associated responses are the tasks that are released simultaneously on arrival.

The schedule records four status variables in addition to the arrivalpoints:

- `State`.
  Records whether the schedule is running or stopped.
- `Next`.
  Records which arrivalpoint will be processed next.
- `Now`.
  Holds the current value of the schedule tick.
- `Match`.
  Holds the tick value at which the next arrivalpoint will be processed.

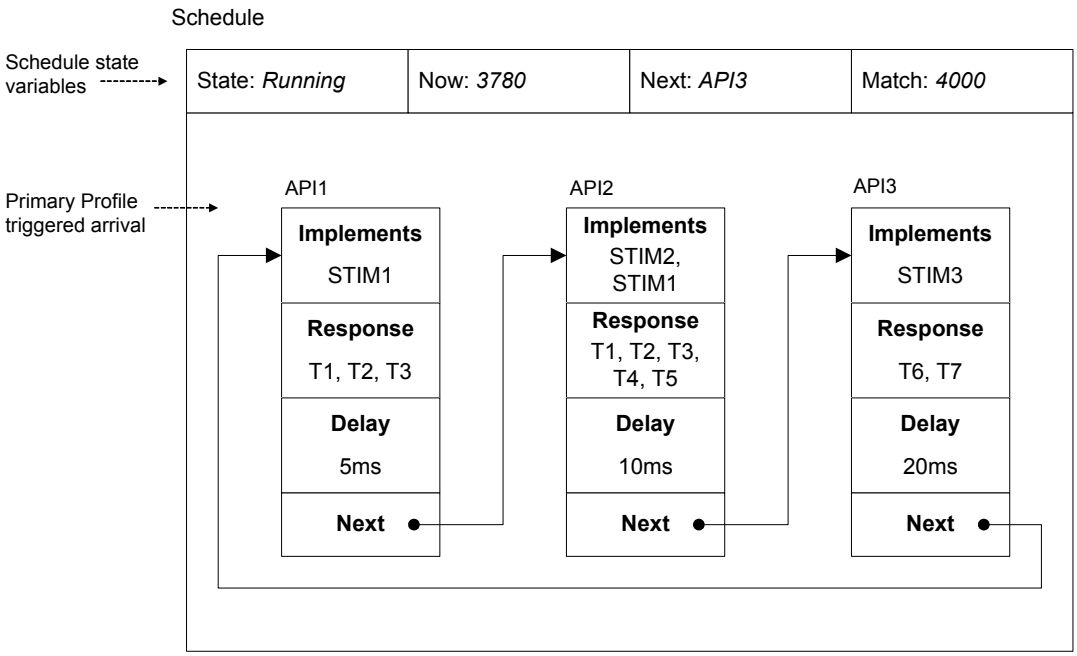


Figure 11-1 - Visualizing a Schedule

11.1.3 Ticked and Advanced Schedules

Schedules can be either:

- Ticked.
  Ticking a schedule is similar to ticking a counter. The schedule maintains an internal count of the number of ticks that have elapsed. It processes the arrivalpoint when the counter value reaches the `match` value.
- Advanced.
  An advanced schedule allows the counter-compare hardware to be exploited on the target hardware. It generates an interrupt to tell the schedule that it must process the next arrivalpoint immediately. This minimizes the number of tick interrupts that will need to be handled. When an advanced schedule is used, an interrupt will only occur when an arrivalpoint needs to be processed.

When you use a ticked or an advanced schedule, you are responsible for providing the **driver**. For a ticked schedule the driver will simply be a periodic interrupt. For an advanced schedule a series of callback routines need to be provided so that RTA-OSEK Component can manage the counter-compare hardware. You can find out more about callback functions in Section 11.11.1.

11.2 Declaring Periodic Schedules

Periodic schedules are declared using the RTA-OSEK GUI. Each schedule must have a unique name and a specified **tick rate** (that can be different in every application mode). You can see from Figure 11-2 how a periodic schedule is declared.

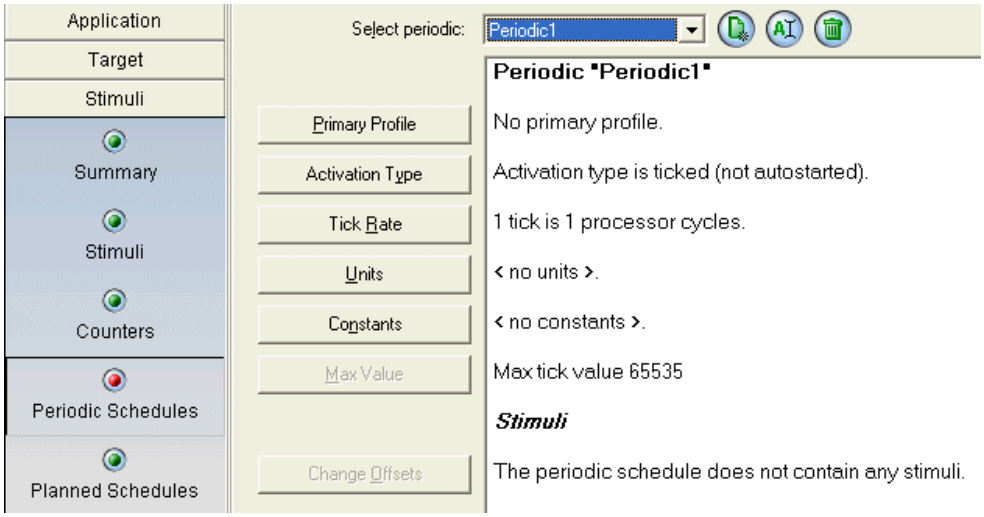


Figure 11-2 - Configuring a Periodic Schedule

A schedule is driven by a primary profile. In your application, this primary profile will usually be an ISR. By default, all periodic schedules are configured as ticked (remember that schedules can be either ticked or advanced). If you

want to change the schedule to be advanced, have a look at Section 11.11 for more information.

Important: The intended behavior of a schedule will only occur if the arrival pattern of the primary profile can be achieved within the resolution of a schedule tick. If you specify an arrival pattern outside the resolution of your schedule, the arrival rate is rounded up to the next arrival pattern achievable with the specified tick rate resolution.

11.3 Building Periodic Schedules

Periodic schedules are built by attaching a periodic stimulus to a periodic schedule. Figure 11-3 shows you how to do this.

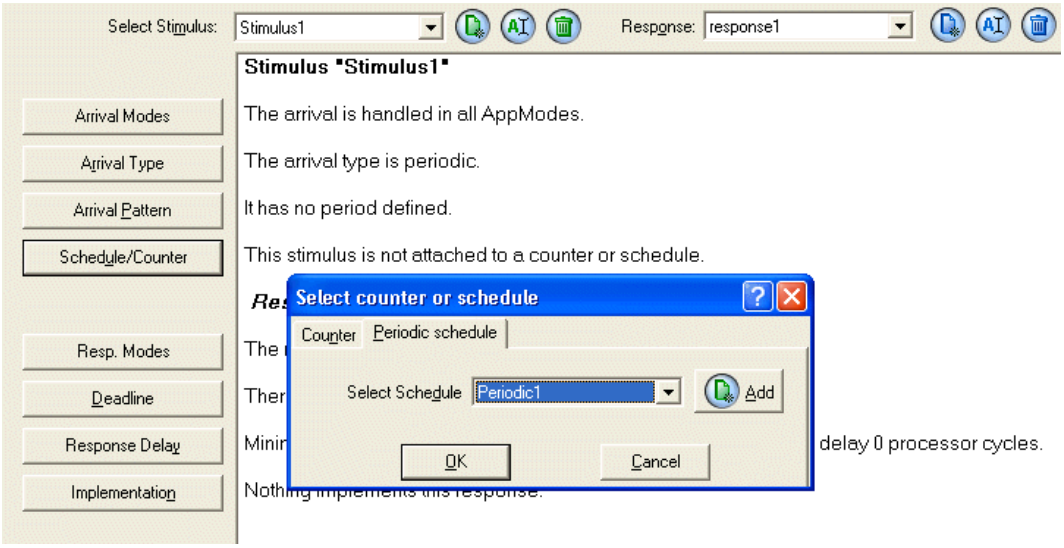


Figure 11-3 - Attaching a Periodic Stimulus to a Periodic Schedule

Attaching a stimulus to the schedule creates an implicit arrivalpoint that can be used by RTA-OSEK at run-time. The arrivalpoint will release all tasks required to generate the responses for the stimuli that it implements on arrival.

The period of the arrivalpoint is taken from the period specified in the stimulus arrival pattern, but is presented in terms of ticks of the schedule.

If, for example, a 20ms stimulus is defined and a tick rate of 1 tick in 5ms is specified, the stimulus will have a schedule period of 4 schedule ticks. This is shown in Figure 11-4.

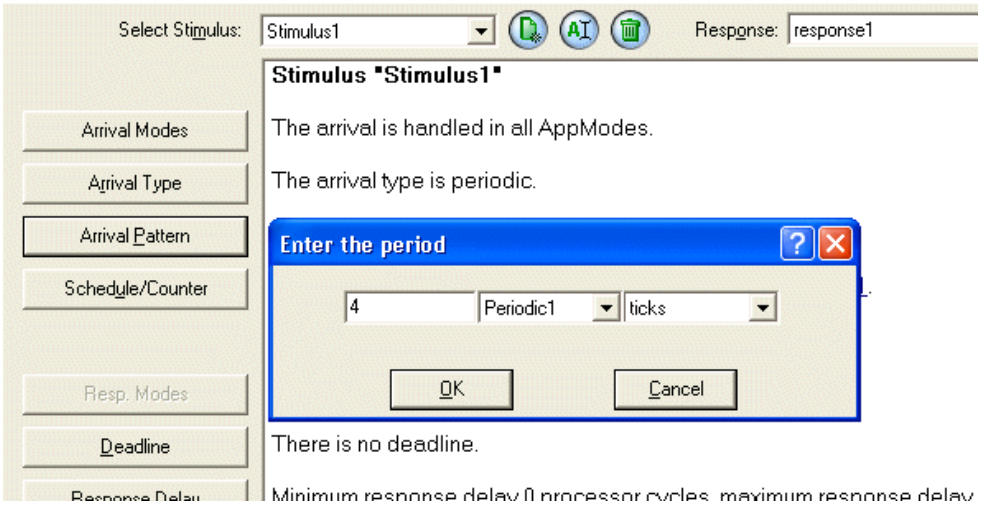


Figure 11-4 - Specifying a Periodic Arrival Pattern

Important: The intended behavior of a schedule will only occur if the arrival pattern of the stimulus can be achieved within the resolution of a schedule tick. If you specify an arrival pattern outside the resolution of your schedule, the arrival rate is rounded up to the next arrival pattern achievable with the specified tick rate resolution.

11.3.1 Visualization of a Periodic Schedule

Schedules can be viewed graphically in RTA-OSEK. You can see this by selecting the **Graphic** tab in the Periodic Schedule workspace. The graphical view will only be available if execution times have been specified for your tasks and ISRs implementing the responses for the stimuli involved.

The visualization will show the arrival of stimuli on the schedule and the primary profile. However, it will not show execution times for stimuli responses, unless you have specified execution profiles.

In Figure 11-5 there are three stimuli, Stimulus1, Stimulus2 and Stimulus3 with periods of 5ms, 10ms and 20ms respectively. These have been attached to a periodic schedule that has a tick rate of 1 tick in 5ms.

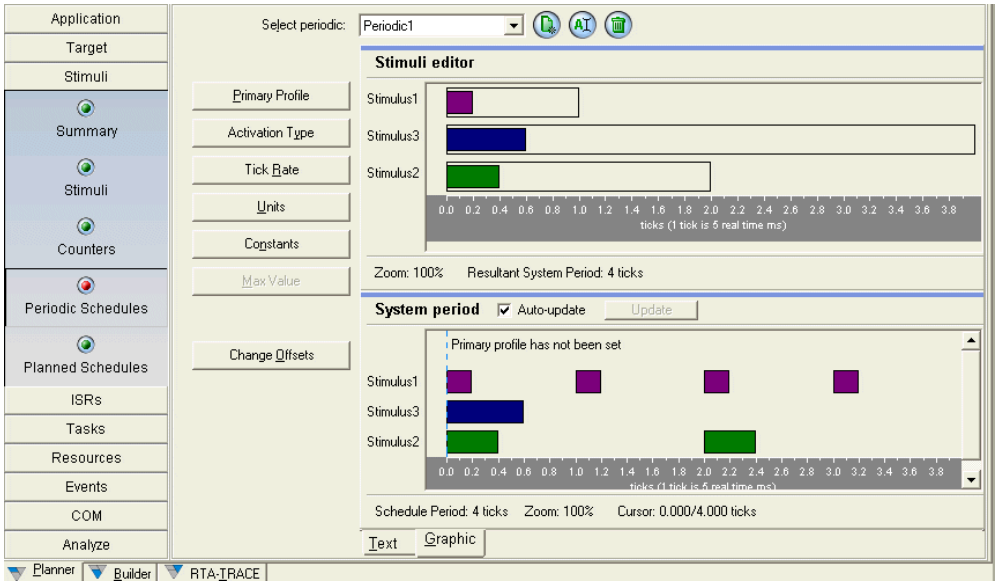


Figure 11-5 - A Graphical Representation of a Periodic Schedule

11.3.2 Editing Periods

To change the period for a stimulus on a periodic schedule, you can modify the stimulus arrival pattern in the Stimulus dialog. You can, however, also do this using the **Stimuli Editor** on the Graphic tab in the Periodic Schedule workspace. The period must be an integer multiple of the schedule tick rate.

Each Stimulus in the Stimuli Editor has a bounding box indicating its period – the right-hand end of this box can be dragged left and right to shorten/extend the stimulus period. Once the period has been changed, the System Period area updates to show the new pattern of execution.

It is also possible to apply an offset to each stimulus (at least one stimulus must have an offset of zero) to even out processor load – in the above example, we see all three stimuli being triggered at time zero; by offsetting stimulus3 by 1 tick, we can ensure that no more than two stimuli occur simultaneously. This may help when confronted with an 'unschedulable' system.

11.4 Declaring Planned Schedules

Systems where the stimuli occur aperiodically can be built using planned schedules.

Each schedule must have a unique name and a specified tick rate. You can see from Figure 11-6 how a planned schedule can be configured.

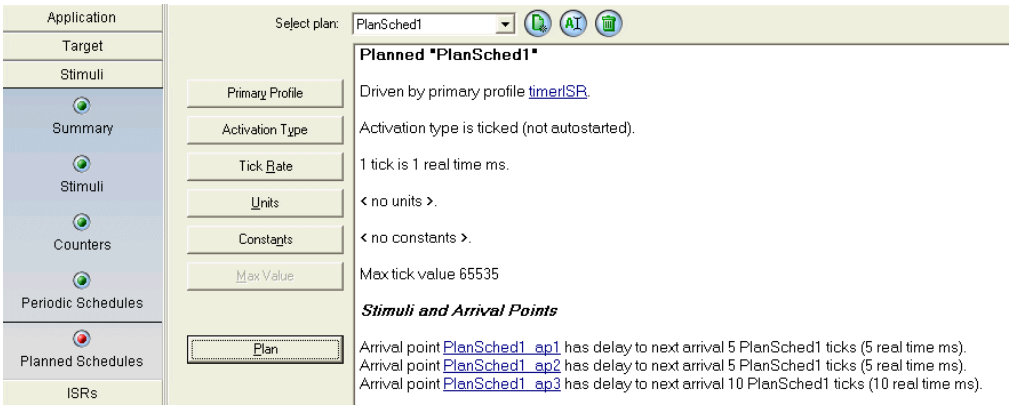


Figure 11-6 - Configuring a Planned Schedule

A schedule is driven by a primary profile. The primary profile is usually an ISR in your application. By default, all planned schedules are configured as ticked. You learnt earlier that schedules could be either ticked or advanced. If you want to change the schedule to be advanced, you can find out how to do this in Section 11.11.

11.5 Building Planned Schedules

When a periodic schedule is built, the stimulus period is used as a specification of the occurrence of the stimuli. For example, a 20ms period implies that a stimulus occurs at 0ms, 20ms, 40ms … and so on.

With a planned schedule a full specification of the arrivals of planned stimuli must be provided (this is why planned stimuli do not have arrival patterns defined in the Stimulus workspace in the same way as periodic stimuli). Timing information is only associated with planned stimuli.

When building a planned schedule you must:

- Attach planned stimuli to the planned schedule.
- Specify which stimuli are attached to which arrivalpoints.

Figure 11-7 shows how planned stimuli are attached to a planned schedule.

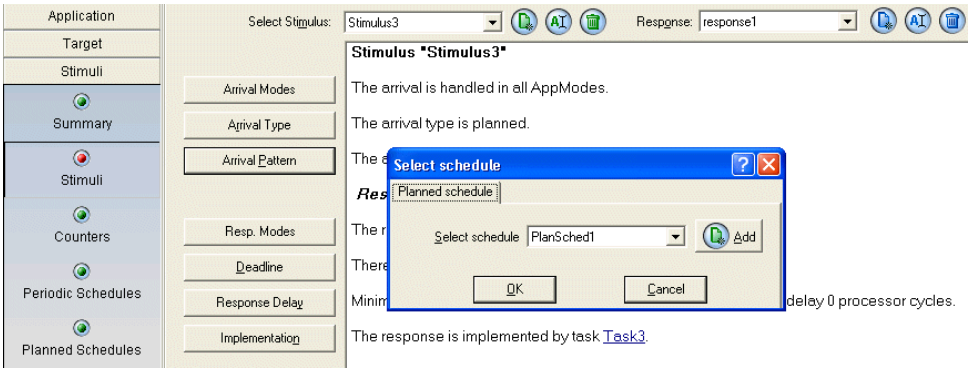


Figure 11-7 - Attaching Planned Stimuli to a Planned Schedule

Each planned schedule has a single plan that contains a set of arrivalpoints. You should use the plan to specify when the stimuli occur. Each arrivalpoint

can contain multiple stimuli and the same stimulus can be attached to more than one arrivalpoint.

11.5.1 Planning Arrivalpoints

Figure 11-8 shows how arrivalpoints are configured.

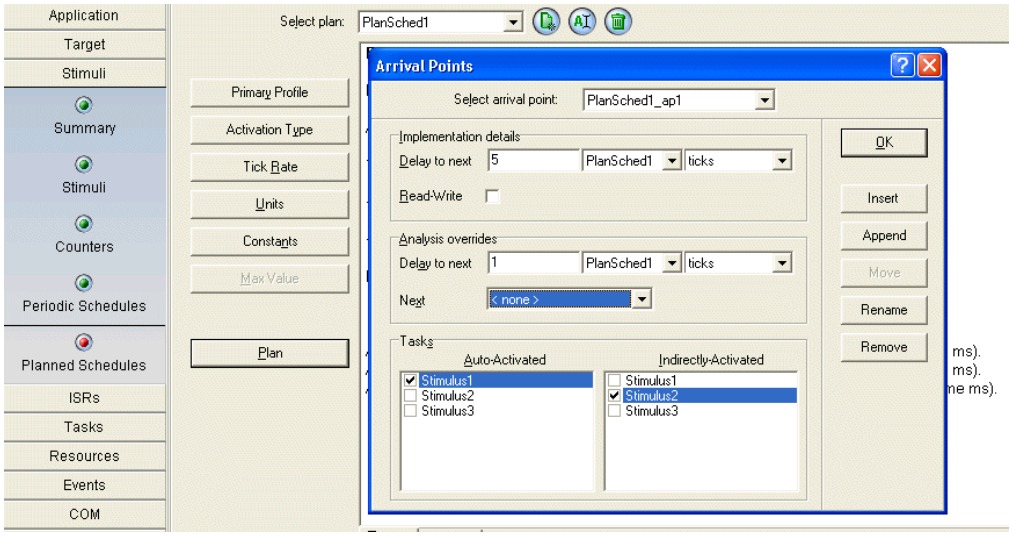


Figure 11-8 - Configuring Arrivalpoints

Each arrivalpoint has:

- A unique name.
- A delay to the next arrivalpoint.
- A set of stimuli that will be triggered on arrival.

For each arrivalpoint you can set the analysis overrides. This is used in timing analysis only.

A plan is created by entering a sequence of arrivalpoints that must be specified at run-time. Configuration of schedules in RTArchitect can be compared with creating a linked-list. Arrivalpoints can be inserted into the schedules or appended to the schedule.

The RTA-OSEK kernel processes arrivalpoints in the order that they are listed. This ordering can be viewed in the workspace.

You can use the dialog in Figure 11-8 to **insert** arrivalpoints before the selected point or to **append** them to the end of the list.

A schedule is **single-shot** if the repeat arrivalpoint is not selected. This means that when the schedule is started it will run to completion then stop. You will find this useful for creating phased sequences of internal stimuli that can be released in response to some sporadic external stimulus (for example, a real-world interrupt).

Planned schedules can be created with loops. The next attribute of the final arrivalpoint in the list can be set to 'point' to any earlier arrivalpoint. To do this a **repeat arrivalpoint** must be specified.

The default minimum delay between arrivalpoints is 1 schedule tick. For most applications this default will need to be modified. For a single-shot periodic schedule, the delay for the final arrivalpoint in the list does not matter.

11.5.2 Attaching Stimuli to Arrivalpoints

Stimuli that must be triggered on arrival are said to be **auto-activated**. These stimuli are selected from the set of available stimuli attached to your schedule. Any number of stimuli can be attached to an arrivalpoint and the same stimulus can be attached to more than one arrivalpoint in your schedule. All stimuli that are attached to your schedule must be attached to at least one arrivalpoint.

In the following example there are 2 stimuli, Stimulus1 and Stimulus2, with the following required arrivals:

- `Stimulus1` must run at 0, 5, 20, 25, 40, 45ms … and so on.
- `Stimulus2` must run every 10ms periodically.

This system can be implemented using a planned schedule with 3 arrivalpoints.

The arrivalpoints are:

- `ap1` auto-activates Stimulus1 and Stimulus2 and has a 5ms delay to `ap2`.
- `ap2` auto-activates Stimulus1 and has a 5ms delay to `ap3`.
- `ap3` auto-activates Stimulus2 and has a 10ms delay to `ap1`.

Figure 11-9 shows how the Planned Schedule workspace will look once the plan has been entered.

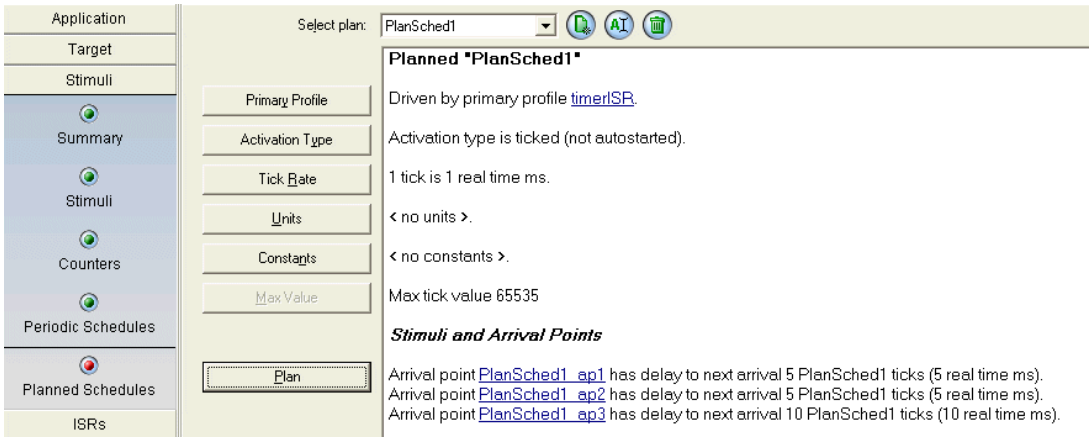


Figure 11-9 - Attaching Stimuli to Arrivalpoints

11.5.3 Visualization of a Planned Schedule

When you have created a plan for a planned schedule you can then view it graphically. This visualization shows stimuli and arrivalpoints. It also shows the execution time information for tasks and ISRs, if this has been specified.

The visualization of the schedule can be seen on the Graphic tab in the Planned Schedule workspace, shown in Figure 11-10. The graphical view shows the arrival of stimuli on the schedule and the primary profile.

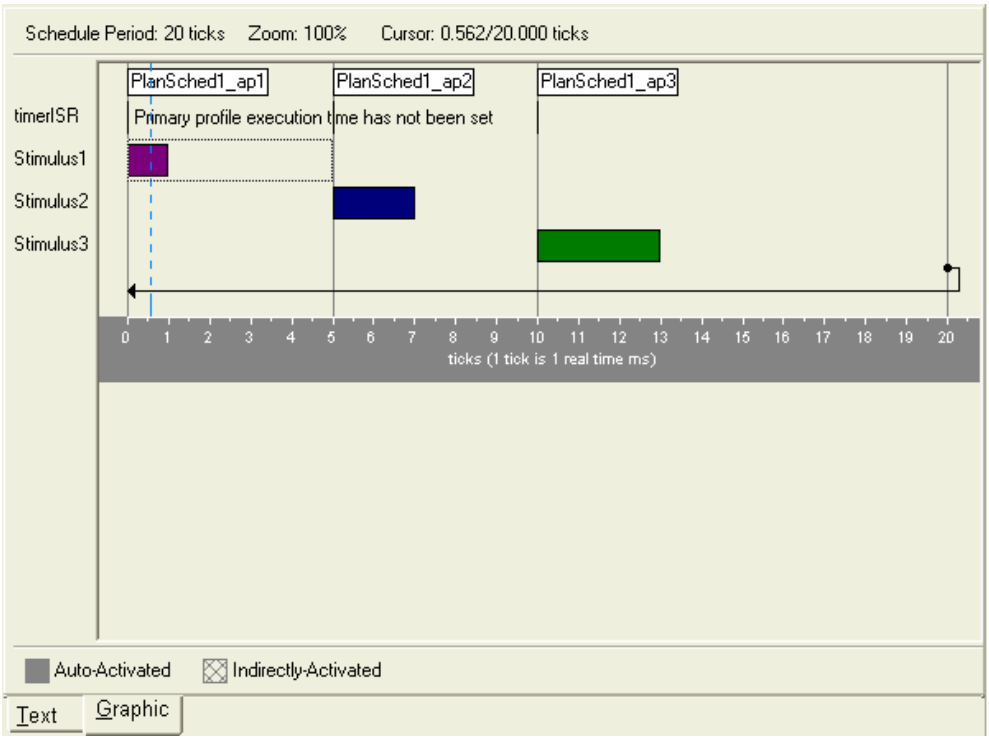


Figure 11-10 - A Graphical Representation of a Planned Schedule

11.5.4 Editing Plans

The planned schedule plan can be edited graphically on the Graphic tab in the Planned Schedule workspace. Moving the mouse over the arrivalpoint, a tooltip appears showing the current delay for the arrivalpoint.

Delays can be changed by dragging the arrivalpoint's time indicator (the vertical bar) left or right. The repeat properties of a planned schedule can also be changed by right-clicking on the arrivalpoint which is to be the start of the repeated sequence, and selecting 'Repeat Arrivalpoint'.

11.6 Starting Schedules

Schedules are started using the `StartSchedule(ScheduleID, TickType)` API call. Schedules are normally started in `OS_MAIN`, but they can be started anywhere in the application.

The `TickType` parameter of the `StartSchedule()` call specifies the relative time from the call being made (in schedule ticks) to the schedule processing the first arrivalpoint. In other words, it sets the `match` value for the first arrivalpoint on the schedule. Code Example 11:1 shows how two schedules can be started.

```
StartSchedule(PeriodicSchedule1, 20);
StartSchedule(PlannedSchedule1, 200);
```

Code Example 11:1 - Starting Schedules

In the first case the `PeriodicSchedule1` is started 20 schedules ticks after the current schedule `now` value. In the second case the `PlannedSchedule1` is started 200 ticks after the current `now` value.

Important: You must make sure that the `match` value that is passed to `StartSchedule()` is sufficiently long, so that it has not already expired before the call returns.

11.7 Ticking Schedules

When a schedule is used in an application you must drive the schedule by providing a tick source. There is no restriction on how a schedule is ticked, but Category 2 ISRs are generally used.

When RTA-OSEK is used to build your application the API call `TickSchedule_<ScheduleID>` is created automatically for each ticked schedule that has been defined. This API call must be made whenever it is necessary to tick the schedule.

If, for example, `Schedule1` and `Schedule2` are both defined in your application as ticked schedules, then RTA-OSEK will generate the following API calls:

```
TickSchedule_Schedule1()
TickSchedule_Schedule2()
```

Code Example 11:2 - TickSchedule() API Calls Generated by RT-OSEK

The interrupt handler that you provide to tick the schedule must call this API. Have a look at Code Example 11:3 to see how a ticked schedule driver is written.

```
ISR(ISR1) {
  ServiceInterrupt();
  TickSchedule_Schedule1();
}
```

Code Example 11:3 - Writing a Ticked Schedule Driver

.

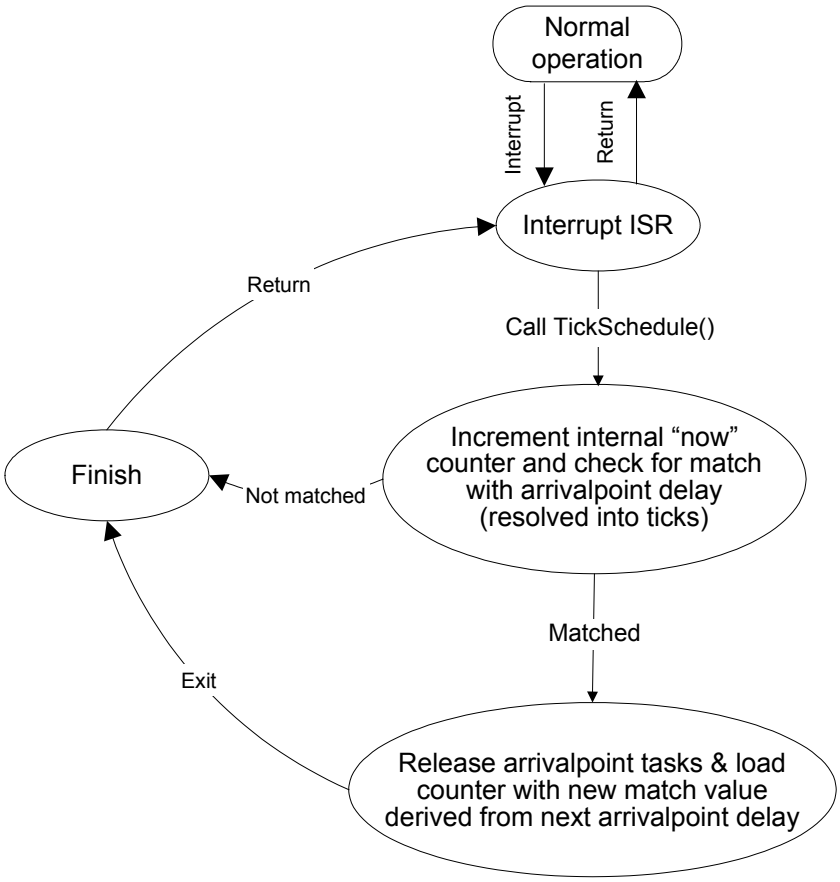


Figure 11-11 - The Ticked Activation State Model

11.8 Stopping Schedules

The `StopSchedule(ScheduleID)` API call can be used at any time to stop a schedule.

This halts the schedule at the current count value. If a single-shot schedule is being used, it will stop automatically after the final arrivalpoint has been processed.

Advanced Scheduling Concepts

11.9 Autostarting Ticked Schedules

Ticked schedules can be configured to autostart a specified number of ticks after `StartOS()` returns.

To start the schedule immediately it should be set to autostart after 1 tick. In this case, the first arrivalpoint will be processed on the next call to `TickSchedule_<ScheduleID>`. A schedule will always start at the first arrivalpoint.

Figure 11-12 shows how a schedule is set to autostart.

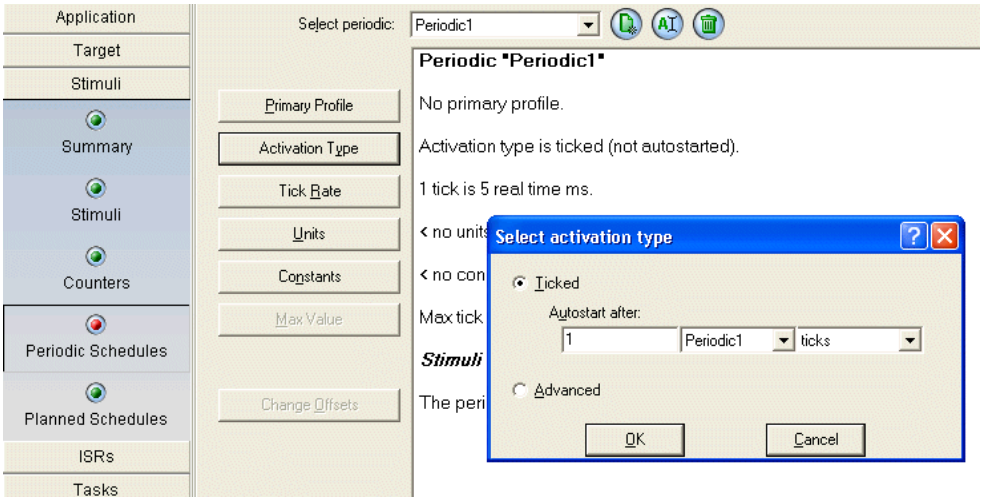


Figure 11-12 - Autostarting a Schedule

11.10 Restarting Single-Shot Schedules

You will need to take special care when you restart a single-shot schedule that has terminated. When this happens, the `next` value of the schedule will be pointing to the last arrivalpoint. To repeat the entire schedule, you will need to use the API call `SetScheduleNext(ScheduleID, ArrivalpointID)` to make sure that the next pointer is reset to the first arrivalpoint in the schedule.

11.11 Advancing Schedules

So far you have looked at ticked schedules, which are useful when arrivalpoint delays have a coarse resolution. The internal counter that RTA-OSEK Component uses to log the current count value has a resolution limited by the size of a `TickType`. You can see the size of a `TickType` in the Target Details in the RTA-OSEK GUI.

If a tick is 1ms, the longest delay that can be specified with a 16-bit `TickType` is 65.535 seconds. If a tick is 1µs, then the longest delay is 65.535 milliseconds.

If you use a ticked schedule you might have to trade off resolution against range. Advanced schedules provide a possible solution to this problem. They allow you to use counter-compare hardware on your target to achieve long ranges at fine resolution*.

Figure 11-13 shows how an advanced schedule operates.

* The scope for doing this depends on the configuration of your target hardware.

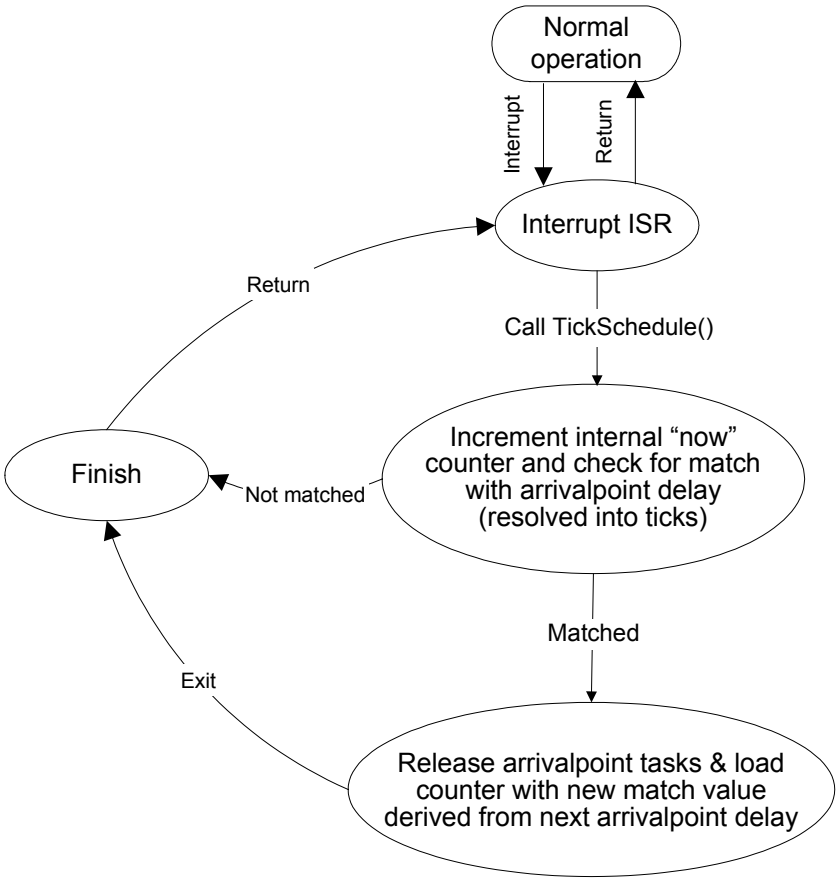


Figure 11-13 - The Advanced Activation State Model

In an advanced schedule, an interrupt is only generated when an arrivalpoint needs to be processed. For example, if a schedule has arrivalpoints at 0, 3 and 7ms and the schedule tick rate is 1ms, then the system is suffering interference from 8 interrupts if the schedule is ticked.

If the schedule is advanced, you will only receive an interrupt for each of the 3 arrivalpoints, reducing the interference from the interrupt by over 50%. This allows you to reduce the amount of interference that your application will suffer due to schedule driver interrupts.

Figure 11-14 and Figure 11-15 show the relative effect of this in visualizations for the ticked and advanced version of this schedule. Figure 11-14 shows a graphical view of the ticked schedule.

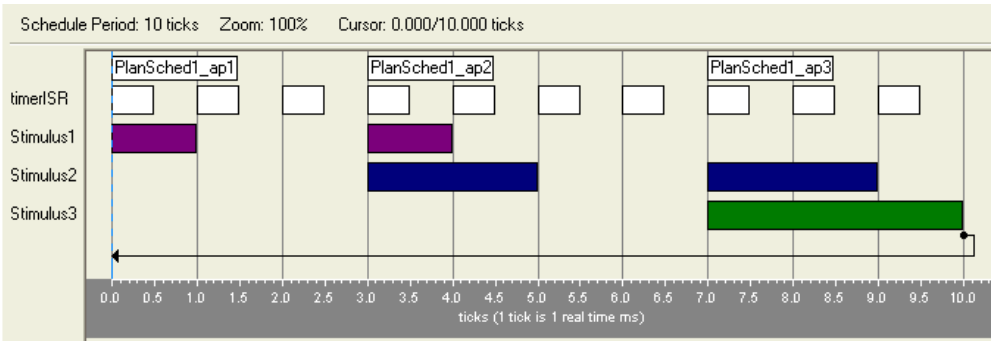

Schedule Period: 10 ticks Zoom: 100% Cursor: 0.000/10.000 ticks
PlanSched1_ap1
PlanSched1_ap2
PlanSched1_ap3
timerISR
Stimulus1
Stimulus2
Stimulus3
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
ticks (1 tick is 1 real time ms)

Figure 11-14 - Ticked Schedule Graphical View

Figure 11-15 shows the advanced version of this schedule. You can see here how the amount of interference has been reduced.

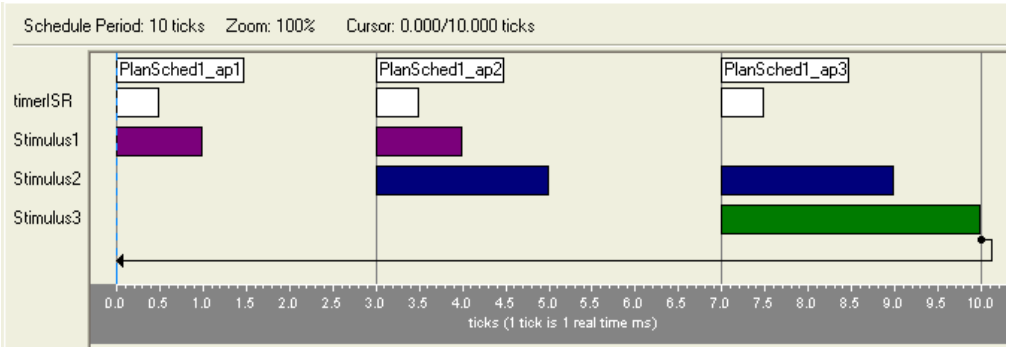

Schedule Period: 10 ticks Zoom: 100% Cursor: 0.000/10.000 ticks
PlanSched1_ap1
PlanSched1_ap2
PlanSched1_ap3
timerISR
Stimulus1
Stimulus2
Stimulus3
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
ticks (1 tick is 1 real time ms)

Figure 11-15 - Advanced Schedule Graphical View

11.11.1 Advanced Schedule Driver Callbacks

For an advanced schedule, RTA-OSEK Component needs to access the counter-compare hardware, so that the next expiry time can be processed. You will need to provide functions that RTA-OSEK Component can use to control this hardware.

Four callback functions must be provided for each advanced schedule. These are:

- `Set_<ScheduleID>`
  Sets up the counter compare hardware. The function prototype is:
  `OS_CALLBACK(void) Set_<ScheduleID>(TickType Match);`
- `State_<ScheduleID>`
  Returns the status of the schedule and the time that the schedule next expires.
  The function prototype is:
  ```
  OS_CALLBACK(void) State_<ScheduleID>
      (ScheduleStatusRefType State);
  ```
- `Now_<ScheduleID>`
  Returns the current value of the counter. The function prototype is:
  `OS_CALLBACK(TickType) Now_<ScheduleID>(void);`

- `Cancel_<ScheduleID>`.
  Cancels any outstanding counter expiry. The function prototype is:
  `OS_CALLBACK(void) Cancel_<ScheduleID>(void);`

The first three of these functions correspond to three of the schedule state variables. The cancel function provides a handle for RTA-OSEK Component to stop the counter. With an advanced schedule this information is maintained by the counter-compare hardware rather than by RTA-OSEK Component. Further information on the Advanced Schedule Driver Interface can be found in the *RTA-OSEK Reference Guide*.

11.12 Using Non-Time Based Schedule Units

Up to now you have seen schedules that use time as the tick. The RTA-OSEK GUI provides a facility to declare schedule **units**. Units allow schedule ticks to be specified in terms of the real-world unit.

You might, for example, have a schedule that counts teeth on a toothed timing wheel and activates tasks at specific angular rotations. A possible abstraction for this would be to declare a degree unit and specify that there are 360 degrees in a revolution. This is shown in Figure 11-16.

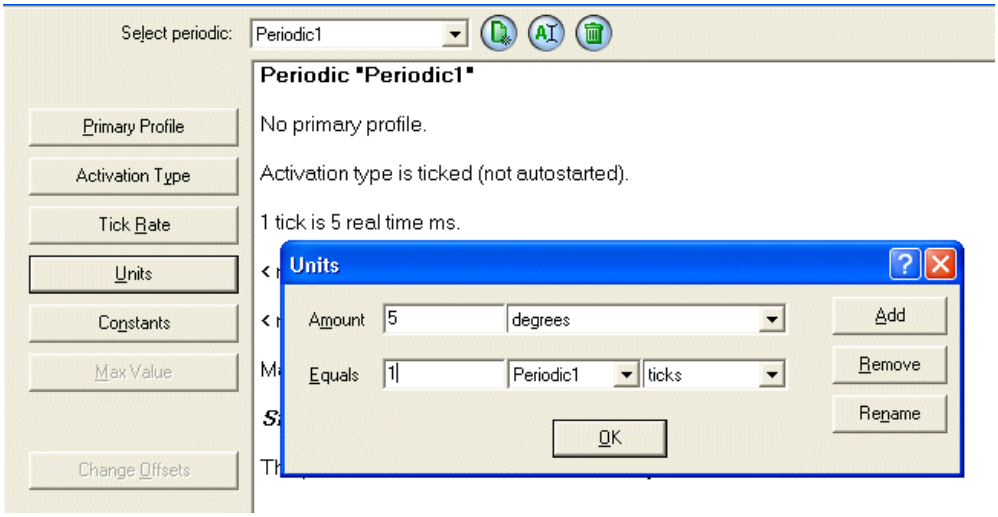


Figure 11-16 - Declaring an Angular Unit

If, for example, an angular schedule uses 1 tick per 5 degrees, you must make sure that your interrupt source provides a tick for each 5 degrees of rotation.

11.13 Specifying Schedule Constants

Delay values can be specified when a schedule is modified at run-time. The delay value is the time that must elapse before the next arrivalpoint is processed. You can declare symbolic **constants** for commonly used delay values.

If you use hard coded numbers in your application, you must ensure that they are scaled appropriately in your application code. If you use constants,

however, you will be able to change values (such as the tick resolution) and the value of the constants will remain correct for your application.

You should use schedule constants wherever possible to define any schedule tick value that you pass into the schedule API calls. The constants are available to your application code through the generated header files.

In Figure 11-17 the constant `Revolution` has been set as 360 degrees.

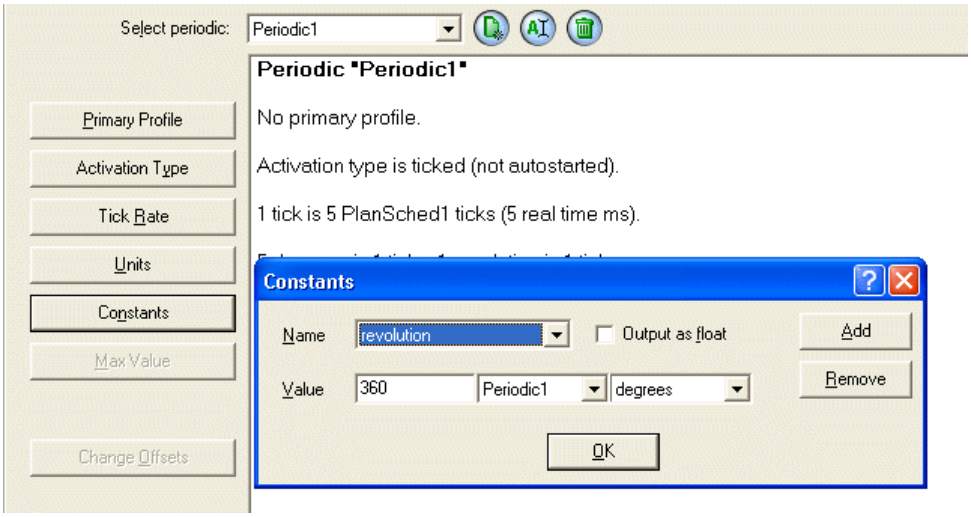


Figure 11-17 - Declaring a Schedule Constant

11.14 Using Periodic Offsets

The behavior within a schedule is constrained. Following the construction of a schedule, you know which tasks will execute at specific times, relative to the schedule itself.

This fact can be exploited in periodic schedules by using **offsets**. Offsets allow you to offset the release of a particular arrivalpoint. The amount of interference and/or blocking suffered by tasks released from other arrivalpoints is, therefore, minimized.

Offsets must be less than the period of the associated stimulus and at least one stimulus in the schedule must have an offset of zero.

Let's look at Figure 11-18. Viewing the schedule graphically, using the visualization that you learnt about in Sections 11.3.1 and 11.5.3, you can see that `Task3` is released at the same time as `Task1` and `Task2`.

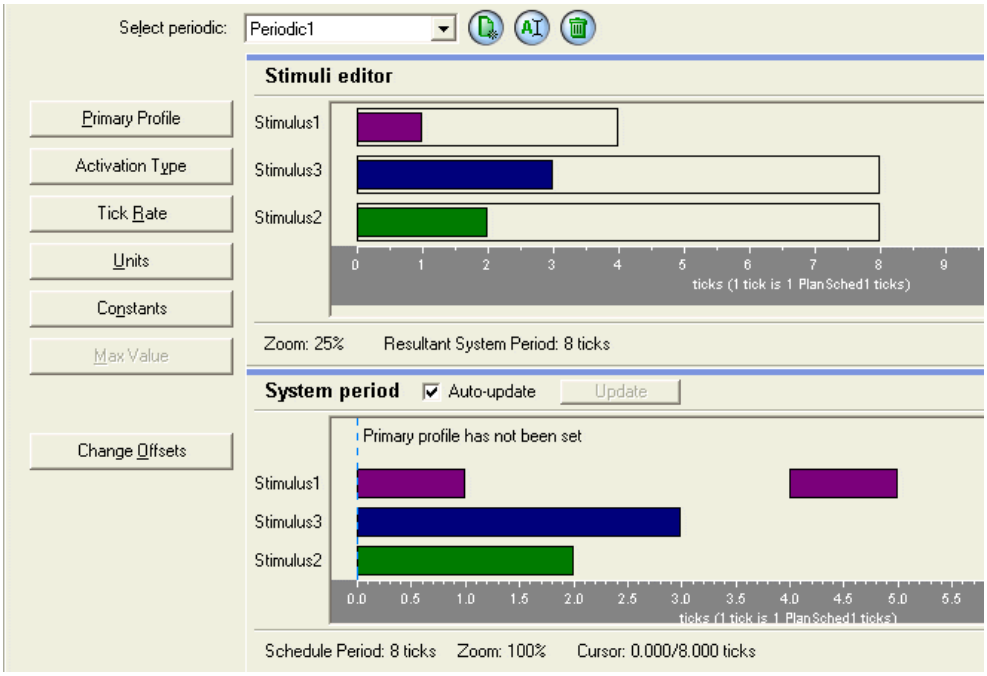


Figure 11-18 - Using Periodic Offsets

The tasks that are released as a result of `Stimulus3` do not get access to the CPU until up to 3ms after release. Once running, they are preempted by a subsequent arrival of `Stimulus1`. So, Figure 11-18 shows that there is a timeframe during which no tasks on this schedule are running. You can see here that the timeframe is longer than the execution time of `Stimulus3`.

By offsetting the release of `Task3` by 5ms (by dragging the left-hand side of the task along to the offset you require), you can remove preemption on the schedule and shorten the response time for the task itself. The effect of modifying the offset is shown in Figure 11-19.

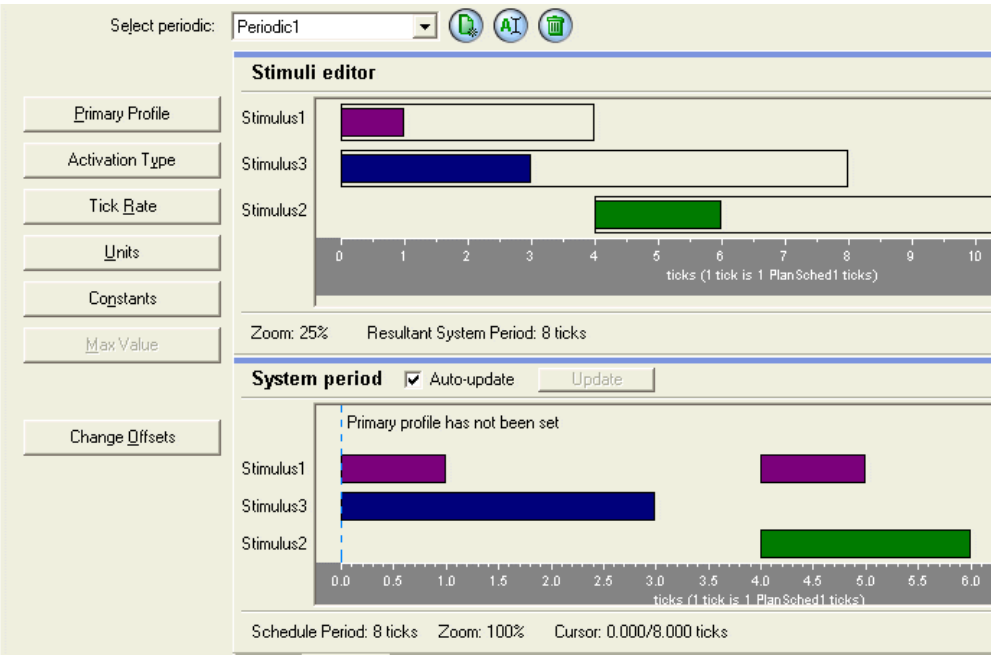


Figure 11-19 - Changing the Offset

Note that other task activations in your application might mean that this offset, whilst better for this single schedule, actually results in a worse performance overall. You can check whether this is the case using timing analysis.

11.15 Modifying Planned Schedules at Run-time

Planned schedules offer a degree of flexibility because the schedule and the associated arrivalpoints can be modified at run-time.

You might be using schedules, for instance, in an engine control application to provide a phased release of tasks where the phasing must change as engine speed increases or decreases.

Alternatively, you might want to provide some run-time fault tolerance. You can do this by allowing a task that reads a sensor value to be replaced with one that synthesizes a value if a fault is detected with the sensor hardware.

RTA-OSEK Component provides API calls to get the current state of the schedule and associated arrivalpoints. API calls are also provided to set the properties to new values.

The status of the schedule is always located in RAM because RTA-OSEK Component needs to update these values at run-time. Arrrivalpoints are located in ROM by default.

Specifying that an arrivalpoint is read-write allows you to modify, at run-time, the taskset that it releases in response to its associated stimuli (you will learn about this in Section 11.15.3). It will also allow you to modify the delay or next attribute at run-time (explained in Sections 11.15.1 and 11.15.2).

Read-write arrivalpoints will be located in RAM. You can set arrivalpoints to be read-write using the RTA-OSEK GUI, as shown in Figure 11-20.

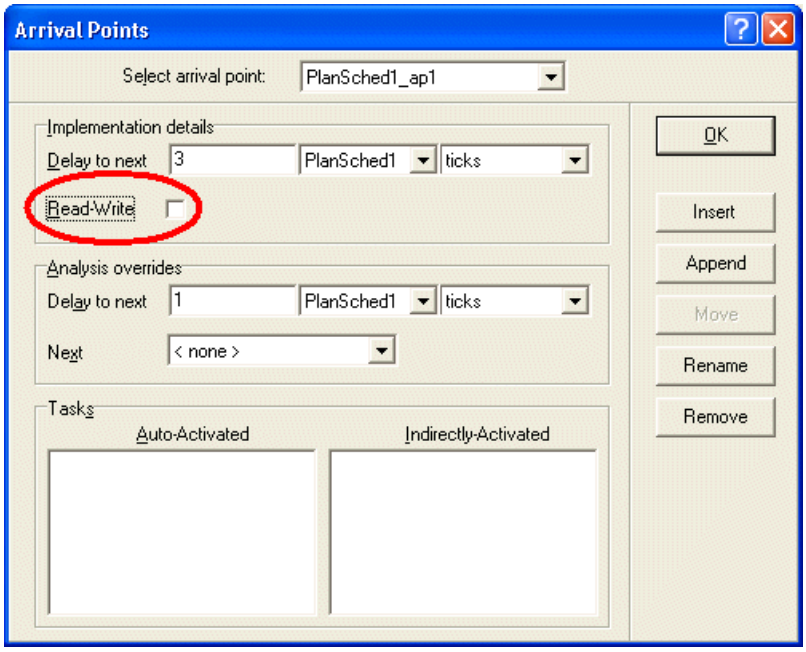


Figure 11-20 - Specifying a Read-Write Arrivalpoint

Important: If you intend to perform timing analysis on your application then you must provide additional details about the worst-case modifications to the schedule that you make.

11.15.1 Modifying Delays

There are two API calls that can be used to modify delay values.

- `GetArrivalpointDelay(ArrivalpointID, TickType)`
  Accesses the current delay value for an arrivalpoint.
- `SetArrivalpointDelay(ArrivalpointID, TickType)`
  Sets the delay values for read-write arrivalpoints.

Important: A delay of zero for the `GetArrivalpointDelay()` and the `SetArrivalpointDelay()` API calls does not indicate a zero delay; it indicates that the delay is the schedule modulus.

11.15.2 Modifying Next Values

Both the schedule and arrivalpoint next values can be changed. If you modify the schedule next value you can change the next arrivalpoint that will be processed.

You will normally edit the schedule next value when you want to make temporary changes to the schedule.

If you want to make permanent changes to the schedule, you should edit the arrivalpoint list so that the changes will continue to exist each time the schedule is processed.

Modifying the next values allows you to create a schedule with optional arrivalpoints. These arrivalpoints can be switched in or out of the schedule at run-time. Any arrivalpoint that you want to switch in, however, must be declared during configuration. An arrivalpoint cannot be dynamically created at run-time.

There are two API calls that can be used to modify next values.

- `SetScheduleNext()`
  Used to modify the schedule next value.
- `SetArrivalpointNext()`
  Used to modify arrivalpoint next attributes at run-time.

In the following example a schedule has three arrivalpoints, `Arrivalpoint1`, `Arrivalpoint2` and `Arrivalpoint3`. In the main program the next arrivalpoint for `Arrivalpoint1` is set to `Arrivalpoint3`.

```
OSMAIN {
  ...
  StartOS(OSDEFAULTAPPMODE);
  SetArrivalpointNext(Arrivalpoint1,Arrivalpoint3);
  ...
}
```

```
TASK(Task1) {
  ...
  /* Switch in the pre-declared Arrivalpoint2.
   * Note that the next for Arrivalpoint2 is
   * already set to Arrivalpoint3 during
   * configuration. */

  SetArrivalpointNext( Arrivalpoint1,
                       Arrivalpoint2 );
  ...
}
```

Code Example 11:4 - Modifying the Arrivalpoint Next Values

If you need to modify the repeat behavior of a schedule, you can set the next arrivalpoint for the repeat to a different arrivalpoint in your application.

11.15.3 Modifying Auto-Activated Tasks

Each arrivalpoint holds a taskset. This taskset represents the tasks that are auto-activated on arrival to generate the responses for the associated stimuli. The taskset is accessible at run-time using `GetArrivalpointTasksetRef()`. If the arrivalpoint is read-write, this API call returns a pointer to a read-write taskset.

The arrivalpoint taskset behaves in the same way as any other taskset in an application, so you can modify the contents at run-time.

Using this feature allows you to dynamically change which tasks are released at run-time. An example is shown in Code Example 11:5.

```
GetArrivalpointTasksetRef( Arrivalpoint1,
                           &TmpTaskset );
MergeTaskset( TmpTaskset, NewTasks );
```

Code Example 11:5 - Modifying Tasks Activated from an Arrivalpoint

11.16 Schedule/Arrivalpoint Tradeoffs

When stimuli are added to a periodic schedule, RTA-OSEK creates implicit arrivalpoints. These arrivalpoints are used by RTA-OSEK Component at run-time to give the specified timing behavior.

Any number of periodic rates can be attached to a periodic schedule. These rates do not need to be harmonic. The number of arrivalpoints created for a periodic schedule, however, is equal to the least common multiple of the individual stimuli periods.

For example, RTA-OSEK will create 3328 arrivalpoints if you create a periodic schedule and attach stimuli with periodic rates of 8ms, 13ms, 16ms, 32ms and 1024ms.

Each arrivalpoint consumes memory. In this example, this method is very wasteful. As an alternative, you could declare two schedules, one for the 1024ms stimulus and one for the remaining stimuli. This solution would require 105 arrivalpoints.

An even better way to do this is to declare a third schedule for the stimulus with the 13ms period. This solution would require 7 arrivalpoints in total.

11.17 Minimizing Schedule RAM Usage

For periodic schedules, state information is maintained in RAM and the (implicit) arrivalpoints are held in ROM.

For a planned schedule you have the option to locate the arrivalpoints in either ROM or RAM. If you need to write to a single arrivalpoint, the rest of the schedule should be located in ROM.

Schedules Summary

- Schedules provide a more flexible alternative to the OSEK counter/alarm mechanism for building complex event-based systems.
- Schedules are not part of the OSEK standard.
- Schedules consist of four state variables and a list of arrivalpoints.
- Arrivalpoints are used to release tasks (or, from an analysis point of view, to implement stimuli) at run-time.
- Arrivalpoints on a schedule are guaranteed to be synchronized at all times.

- On arrival at an arrivalpoint, RTA-OSEK Component will activate the set of tasks required to generate responses to stimuli that the arrivalpoint implements.
- Periodic schedules offer a shorthand way of specifying periodic behavior. All arrivalpoints are implicit and are only used internally by RTA-OSEK Component.
- Planned schedules require an explicit plan of the schedule timing characteristics to be created.
- Planned schedules can be modified at run-time to cater for special system behavior.

12 Startup and Shutdown

Some operating systems that you might have used before will take control of the hardware. RTA-OSEK Component, however, is different.

Initially the operating system is not running, so you are free to use the hardware as if no real-time operating system is being used. Until you explicitly start the operating system with an API call, it is not running.

RTA-OSEK Component can be started in different **modes**. A mode is a set or subset of the complete application functionality that corresponds with a specific function of the application. You will learn more about application modes in Section 12.2.2.

12.1 From System Reset to `StartOS()`

This section looks at what has to be done between an embedded processor "coming into life" when power is applied and the `StartOS()` API call being made to start RTA-OSEK Component and your application. The details of what goes on in this period are naturally dependent on the particular embedded processor in use –the underlying principles are however the same. You should read this section in conjunction with the reference manual for your target processor and apply the concepts we describe to your own platform.

12.1.1 Power-on or Reset to `main()`

When power is applied to an embedded processor, or the processor is reset, the processor does one of two things (depending on the type of processor).

It may start executing code from a fixed location in memory, or it may read an address from a fixed location in memory and then start executing from this address. The fixed location in memory that contains the address of the first instruction to execute is often called the "reset vector" and is sometimes an entry in the interrupt vector table.

In a production environment the reset vector and/or the first instruction to be executed is usually in non-volatile memory of some variety. In a development environment it is often in RAM to permit easy re-programming of the embedded processor. Some evaluation boards (EVBs) have switches or jumpers that permit the reset vector and/or the first instruction to be in EEPROM or RAM.

Going from power-on or reset to the first instruction being executed is often referred to as "coming out of reset". After a processor has come out of reset it usually:

- has interrupts disabled,
- is in supervisor mode (if the processor supports it) - i.e. it can execute all instructions and access all addresses without causing an exception and has all forms of memory and I/O protection turned off.

- is in single-chip mode (if the processor supports it) – i.e. the chip is in a "self-contained mode" where external memory is not usable and external buses are disabled.

It is possible to have any code you like executed when a processor comes out of reset but it is normal if using a high-level language such as C for this bootstrap code supplied with the compiler. The compiler vendor supplies an object module or library that contains the bootstrap code. The bootstrap code carries out basic processor configuration – e.g. bus configuration and enabling of access to internal RAM – and then invokes the C language start-up code – most of this is concerned with initializing data structures, clearing memory, setting up the stack pointer, etc.

Directives in the object module/library or in the linker configuration file are used to ensure that the bootstrap code (and reset vector value if needed) are put in the correct place in memory.

C Language Start-up Code

The C language start-up code is either supplied by the compiler vendor or (on some platforms, in a slightly modified version) by LiveDevices. The start-up code is often supplied in an object module with a name like "`crt0`" or "`startup`" and the code can usually be identified in a map file by looking for a symbol with a name something like "`_start`" or "`__main`". The source to this module is usually available to the user.

On some platforms we supply a different version of the standard startup code that should be used with RTA-OSEK applications. The RTA-OSEK Binding Manual and the example supplied with RTA-OSEK will tell you how to use this.

The start-up code initializes the C language environment. For example it sets up the stack pointer, the heap used for `malloc()` and it initializes global variables by copying their default values from ROM into RAM. Finally the start-up code invokes the application start-up code.

12.1.2 The Application Start-up Code

The application start-up code is the function called "`main()`" or in an RTA-OSEK application the function declared with the macro "`OS_MAIN()`". The application start-up function has three things to do in an RTA-OSEK application:

- Initialize the target hardware into a state where RTA-OSEK and the application can run
- Call `StartOS()` to start RTA-OSEK Component running.
- Carry out idle-task processing.

For example the application start-up code for an RTA-OSEK application may look like:

```
OS_MAIN()
/* note that we use this macro for portability
rather than main() as some compilers expect strange
declarations of main() */
{
  init_target();

  StartOS(OSDEFAULTAPPMODE);

  /* Code that makes up the idle task */
  /* functionality. */

  /* The idle task must never terminate so if */
  /* there is no idle task functionality then */
  /* use something like: */

  for (;;) { /* Do nothing. */ }
}
```

Figure 12:1 : A Typical Main and Idle Task

`StartOS()` starts RTA-OSEK Component running. Once the kernel is running ISRs will be called in response to interrupts occurring and tasks will be scheduled. When `StartOS()` returns the application start-up code is running as the idle task. The idle task must never terminate so if there is nothing for the idle task to do an infinite loop must be used.

The `init_target()` function in the above example is supplied by the user and is used to initialize the target hardware. The remainder of this section describes the types of things that you may have to do to initialize target hardware into a state where your application and RTA-OSEK Component can run. This description is necessarily generic as every embedded processor is slightly different. It is probably wise to read this section in conjunction with the RTA-OSEK Binding Manual for your processor and the processor's reference guide.

A Note on the Startup Hook

If enabled – using "Application / OS Configuration" in the RTA-OSEK configuration tool – the `StartOS()` function will call the startup hook after it has initialized RTA-OSEK Component but before it lowers the interrupt priority level to user level and schedules any tasks. This feature can be used to carry out the final stages of target initialization – see the section on interrupts below. The startup hook is an application provided function called `StartupHook()`.

Setting up Memory

In general memory configuration is carried out by the bootstrap code that is run before the application start-up code is executed. In more complex

embedded processors, however, the memory configuration set-up by the bootstrap code may not be what is required for the application. For example, if the processor has internal RAM and an external memory bus it is most likely that the bootstrap code will have configured the processor to use the internal RAM. If your application needs to use RAM on the external memory bus then you will need to configure the processor to use the external RAM. Configuring access to RAM typically involves programming bank select and mask registers – however the details depend on the embedded processor.

Setting up Peripherals

Most embedded applications make use of peripheral devices which may be part of the embedded processor or attached through I/O or memory buses. Examples are CAN controllers, Ethernet controllers and UARTs. It is generally a good idea to set-up peripheral devices before RTA-OSEK Component is started since at this point the application code cannot be pre-empted and has complete control over interrupts.

Setting up Timers

Most embedded applications use hardware timers. Timers are usually configured to "tick" and generate interrupts at a fixed frequency. The ISR associated with the timer interrupts then either activates a task directly or ticks an OSEK counter (i.e. calls `Tick_xxxx()` where xxxx is the name of the counter).

Setting up a hardware timer depends on the design of the timer but there are two common forms. In the first, a count register is set to zero and a bound register is set to the maximum value for the count register. The count register is incremented by the processor at a given frequency and when it reaches the value in the bound register it generates an interrupt and resets the count register to 0. In the second form a count register is loaded with the number of ticks to occur before an interrupt should be generated. The processor decrements the count register at a given frequency. When the register reaches zero an interrupt is generated. Usually the ISR that handles the interrupt is responsible for reloading the count register.

The frequency at which timers must run will depend on your application. It is vital that OSEK counters are ticked at the frequency specified in their definition.

In extended and timing builds of RTA-OSEK applications a callback function called `GetStopwatch()` must be supplied that returns the value of a free running timer that is incremented at the frequency specified via the RTA-OSEK configuration tool under "Target / Timing Data". See the "Execution Time" section of the RTA-OSEK Reference Guide for details.

You will also need to set-up timers to drive Advanced Schedules. See the section on Advanced Schedules in the RTA-OSEK Reference Guide for details of what must be done.

Setting up Interrupts

Interrupt sources for category 1 and 2 interrupts should be configured before `StartOS()` is called. Category 1 interrupts may also be enabled so that they generate interrupts immediately as the handling of category 1 interrupts is completely outside the scope of RTA-OSEK Component. Category 2 interrupt sources must not actually generate interrupts until after `StartOS()` has completed initialization.

Set-up the category 2 interrupt sources before calling `StartOS()` and then enable actual generation of interrupts in the `StartupHook()` function called by `StartOS()`. `StartOS()` raises the interrupt priority level (IPL) to OS level as soon as it is called and lowers it to user level just before it returns. Thus enabling interrupt generation in `StartupHook()` will not actually result in an interrupt occurring until `StartOS()` lowers the IPL just before it returns.

Ensure that the IPL is set to OS level and then both configure interrupt sources and enable interrupts. Interrupts will not actually be generated until `StartOS()` lowers the IPL just before it returns.

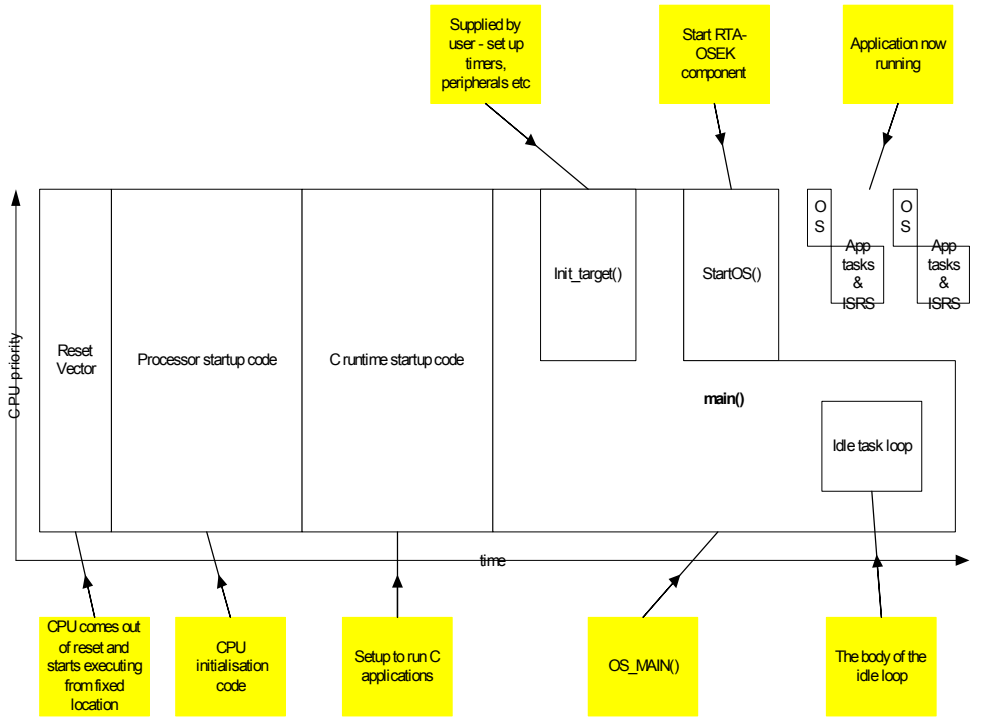


Figure 12:2: System Startup

12.1.3 Memory Images and Linker Files

When you build your application, the various pieces of code, data, ROM and RAM must be located at the right place in memory. This is typically done by the *linker* which resolves references made by user-supplied code to the RTA-OSEK Component library, binds together the relevant object modules and allocates the resultant code and data to addresses in memory before producing an image that can be loaded onto the target.

But how does the linker know what to put where in memory? How does it know where to find ROM and RAM, for example, and what must be allocated to each of them?

Sections

Code and data output by compilers and assemblers is typically organized into "sections", with each "section". You might see a piece of assembler that says something like

```
.section CODE
.public MYPROC
mov          r1, FRED
add          r1, r1
ret
.end CODE
.section DATA
.public FRED
.word 100, 200, 300, 400
.end DATA
.section BSS
.public WORKSPACE
.space 200
.end BSS
```

Figure 12:3: Example Assembler Output Showing Sections

This means that the code for MYPROC should be assembled and the object code should assume that it will be located in a section of memory called "CODE" whose location we will later define in the linker. Similarly, the data labeled "FRED" will be placed in a section called "DATA", and a space of 200 bytes labeled "WORKSPACE" allocated in section "BSS".

C compilers typically output your code into a section called "code" or "text", constants that must go into ROM in a section called something like "const", and variables into "data". There will usually be more – consult the reference manual for your toolchain for more details on what the sections are called and familiarize yourself with where they need to go.

RTA-OSEK itself uses several sections that must be correctly located.

Section	ROM/RAM	Description
os_vectbl	ROM	The interrupt vector table, if generated by RTA-OSEK.
os_pur	RAM	RTA-OSEK uninitialized data
os_pid	ROM	RTA-OSEK read-only data

Section	ROM/RAM	Description
os_pir	RAM	RAM data used by RTA-OSEK that must be initialized at runtime – the initializer for this is in os_pird and it will be initialized by the StartOS() API.
os_pird	ROM	The initializer for os_pir
The following two sections may be used on some platforms that support separate "near" and "far" address spaces (see below)		
os_pnir	RAM	RAM data used by RTA-OSEK that must be initialized at runtime – the initializer for this is in os_pird and it will be initialized by the StartOS() API.
os_pnird	ROM	The initializer for os_pnir

So far we have yet to map these onto addresses in "real" memory. We must therefore look at how these sections are mapped into a memory image.

"Near" and "Far" space

On some processors there exist regions of memory space that can be addressed economically (typically with shorter, smaller instructions that have simpler effective-address calculations), are located on-chip rather than off-chip, or that are fabricated in a technology such that they are more cycle-efficient to access. RTA-OSEK terms this memory "near" space and on these processors places some key data in these areas. On such platforms you will be supplied with information on where you must locate "near" space in ROM and/or RAM, and told in the binding manual what data is placed in it. "Far" space refers to the whole of memory.

Program and Data Space on Harvard Architectures

Most of the discussion about memory so far has assumed the conventional "von Neumann" architecture, in which data and code occupy one address space with ROM and RAM located at different offsets inside this. Some processors (typically very small microcontrollers like PICs, or high-performance Digital Signal Processors) adopt a "Harvard" architecture, in which there are distinct address spaces for code and data (there are some performance advantages to this that offset the programming disadvantages). On a Harvard-architecture processor, RTA may use data space (typically RAM) to store data that would normally be ROM constants on a von Neumann architecture processor, and the startup code will typically contain code to fetch the a copy of the constant data into data space. If you are using a Harvard architecture

processor, the RTA-OSEK binding manual will contain information on any use of RAM to store copies of constants.

The Linker Control File

The linker control file governs the placement of code, data and reserved space in the image that is downloaded to the target microcontroller. Linker files vary considerably between platforms and targets, but typically include at least the following:

- declarations of where ROM and RAM are located on chip – these vary across different variants in a CPU family.
- Lists of sections that can be placed into each memory space
- Initialization of the stack pointer, reset address, interrupt vectors etc.

Let us examine a hypothetical linker control file:

```
ONCHIPRAM start 0x0000 {
  Section .stack size 0x200 align 16 # system stack
  Section .sdata align 16 # small data
  Section os_pnir align 16 # RTA near data
}

def __SP = start stack    # initialize stack ptr

RAM start 0x4000 {
  Section .data align 16  # compiler data
  Section .bss align 16   # compiler BSS
  Section os_pur align 16 # RTA zeroed RAM
  Section os_pir align 16 # RTA initialized RAM
}

ROM start 0x8000 {
  Section .text             # compiler code
  Section .const            # compiler constants
  Section os_pid align 16   # RTA data
  Section os_pird align 16  # RTA initializer
  Section os_pnird align 16 # RTA initializer
}

VECTBL start 0xFF00 {
  Section os_vectbl          # RTA vector table
}

def __RESET = __main         # reset to __main
```

Figure 12:4 A Linker Control File

The file above defines four separate parts of memory – “ONCHIPRAM”, “RAM”, “ROM”, and “VECTBL”. Into each section are placed the appropriate data, as described by the comments.

The example application supplied with RTA-OSEK will contain a fully-commented linker control file; consult this and the Binding Manual for details of how to locate the sections correctly for your target platform.

12.1.4 Downloading to your Target

The output of the linker is typically a binary file in some well-known format (e.g. *a.out*, *coff*, *elf* or *IEEE695*). These can typically be read by debuggers, in-circuit emulators or in-circuit programming equipment, although in some cases it is necessary to convert the output from this binary format into a text-based form (such as *S-Records* or *Intel Hex*) that can be transmitted to a simple boot monitor on the target over a serial link. Tools to do this are usually supplied with your development environment. Consult the documentation on your target platform and development toolchain for details of how to program applications into non-volatile memory.

12.1.5 ROMability

All ports of RTA-OSEK are ROMable and are tested running on a target CPU without any debugger or development equipment connected.

12.2 Starting RTA-OSEK Component

RTA-OSEK Component is started only when a `StartOS()` call is made. This call is usually made from `main()`*. It is up to you to perform any hardware initialization that is necessary for the application. The initial state of RTA-OSEK Component is described in the *RTA-OSEK Reference Guide*.

`StartOS(Appmode)` takes a single application mode parameter. This parameter is either the default mode `OSDEFAULTAPPMODE` or another mode that has been configured in the RTA-OSEK GUI.

Have a look at the example main function in Code Example 12:1, which starts the operating system in the default application mode.

```
#include "osekmain.h"

OS_MAIN(main)
{
  InitializeTarget();

  StartOS(OSDEFAULTAPPMODE);

  for (;;) {
    /* Idle task. */
  }
}
```

Code Example 12:1 - Example Main Function

* RTA-OSEK applications tend to use OS_MAIN() rather than main(). This is so that applications are portable.

When the call `StartOS()` returns, RTA-OSEK Component is running and all interrupts are enabled. Code that appears after `StartOS()`, in the calling function, is treated as the idle task.

Remember that the idle task is just like any other task except that it can never terminate. If you do not want RTA-OSEK Component to terminate, you must make sure that the idle task is an infinite loop.

Most RTA-OSEK Component API calls can be made from the idle task. However, you cannot use any calls that require the idle task to terminate. If you want to find out more about these API calls, have a look at the *RTA-OSEK Reference Guide*.

Important: RTA-OSEK Component API calls cannot be made and Category 2 interrupts are not handled before a call to `StartOS(Appmode)` has returned.

RTA-OSEK Component can be suspended by disabling all Category 2 interrupts and ensuring that they will not be raised on some future event, such as an output compare match.

RTA-OSEK Component will be suspended when no Category 2 interrupts are raised and the idle task is *running*. You can resume RTA-OSEK Component by re-enabling Category 2 interrupts and then resume making RTA-OSEK Component calls.

Important: It is not possible to restart RTA-OSEK Component once it is running. You can only restart the operating system by performing a processor reset.

12.2.1 Shutting Down RTA-OSEK Component

The operating system can be shutdown at any point by making the `ShutdownOS()` API call. When this happens, RTA-OSEK Component will immediately disable interrupts and then enter an infinite loop. If you have configured the `ShutdownHook()` it is called before the infinite loop is entered.

Advanced Startup and Shutdown Concepts

12.2.2 Application Modes

OSEK provides **application modes**. These allow you to control which tasks and alarms are automatically started when the operating system starts (and also allow you to specify different timing behaviors for the system in each mode).

Applications can be started in different modes, which are part of the complete functionality. These modes correspond with specific functions of the

application. You could have, for example, an end-of-line programming mode, a transport mode and a normal mode.

`OSDEFAULTAPPMODE` is the default application mode. You can define as many application modes as you want using the RTA-OSEK GUI. You can see, from Figure 12:5, how they are added to an application.

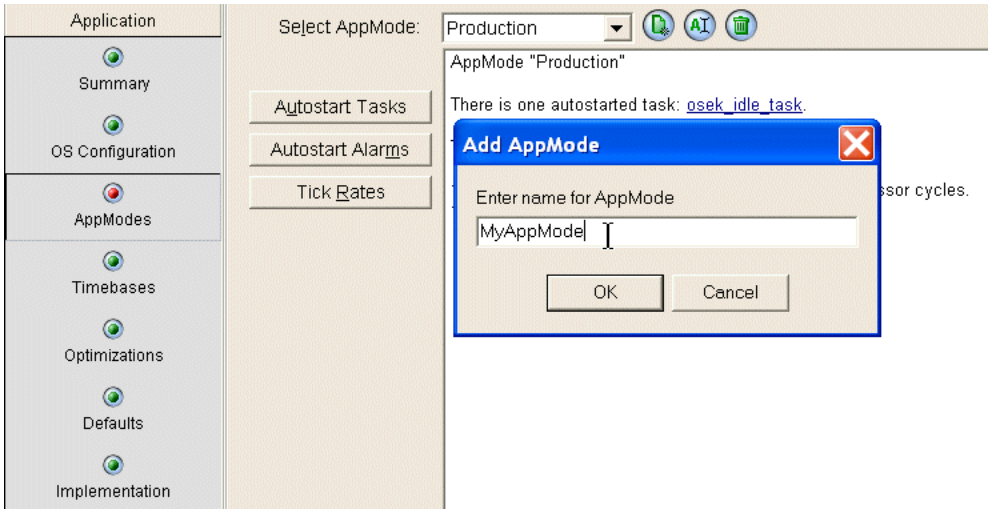


Figure 12:5 - Configuring Application Modes

Important: You can only select application modes before starting the operating system. Once RTA-OSEK Component is running, you cannot change the mode.

`StartOS(Appmode)` will activate any tasks and set any alarms that you have specified to be autostarted.

12.2.3 Autostarting Tasks

The RTA-OSEK GUI is used to set tasks to autostart during a call to `StartOS()`. Figure 12:6 shows how the task autostart options are set.

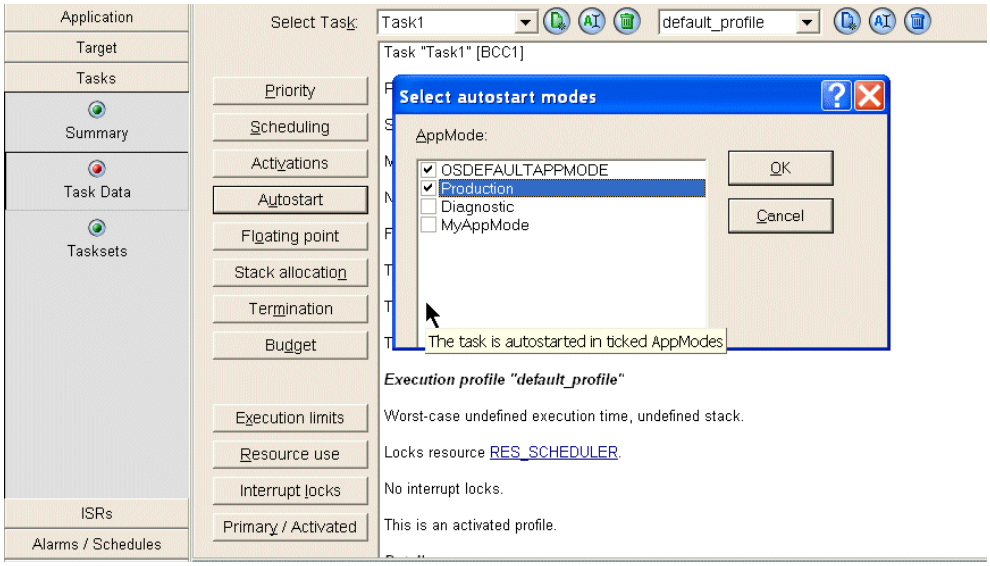


Figure 12:6 - Declaring an Autostarted Task

You can specify that autostarting occurs in whichever application modes you choose. All of the autostarted tasks will have run when `StartOS()` returns. In this case `Task1` has been autostarted in `OSEKDEFAULTAPPMODE` and `Production` application modes.

12.2.4 Autostarting Alarms

Alarms can be autostarted in the RTA-OSEK GUI. When `StartOS()` returns, all autostarted alarms will have been enabled. Figure 12:7 shows you how an alarm is set to autostart.

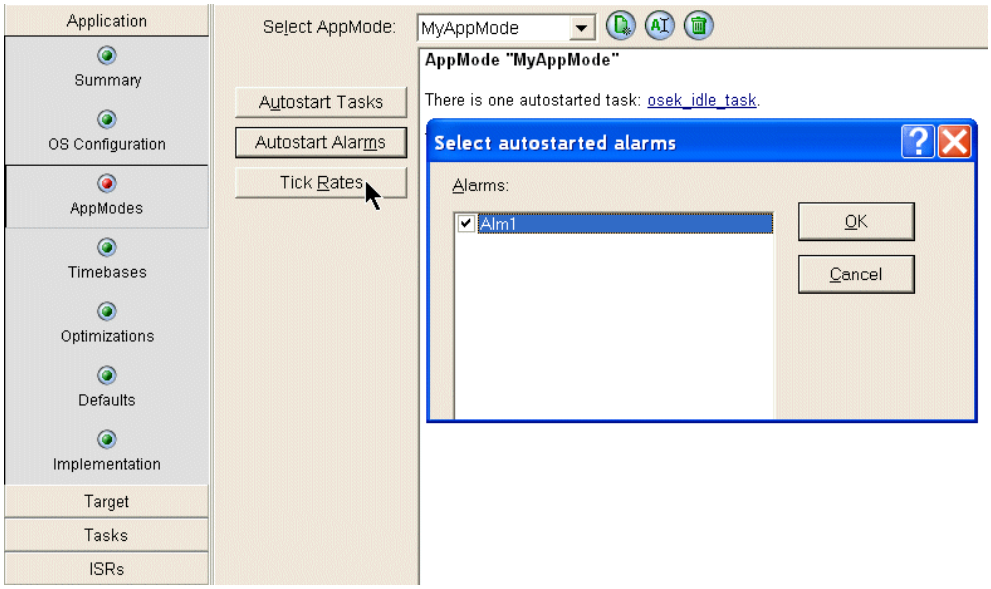


Figure 12:7 - Autostarting an Alarm

Alarms are autostarted through the application modes pane, so you are in effect selecting which alarms are autostarted on a per-application-mode basis.

The alarm in Figure 12:7 has been set to autostart in the `MyAppMode` application modes.

If you want a number of alarms to be synchronized at run-time, then you must make sure that the alarms are autostarted. This is the only way to guarantee alarm synchronization.

Startup and Shutdown Summary

- RTA-OSEK will not work unless everything is located in the right place in memory.
- There are several operations that must be carried out before RTA-OSEK Component can run.
- RTA-OSEK Component doesn't run until the `StartOS()` call is made.
- RTA-OSEK Component can be stopped at any time using the `ShutdownOS()` call.
- Application modes allow you to control the tasks and alarms that are autostarted.

13 Error Handling and Execution Monitoring

During the early stages of development you will need to debug and monitor the execution of your application. Execution monitoring can be as straightforward as generating a trace of the tasks as they run. You might, however, need to monitor the actual execution time or stack usage of tasks to obtain worst-case values for timing and stack analysis.

RTA-OSEK provides OSEK hooks. A hook is a user provided C function with a specified API. The hooks are called by RTA-OSEK Component at particular points during its operation.

Code that runs inside a hook function can make a restricted number of API calls. The *RTA-OSEK Reference Guide* lists these restrictions.

OSEK defines the following hooks:

- Startup Hook.
- Shutdown Hook.
- Error Hook.
- PreTask Hook.
- PostTask Hook.

The OSEK hook routines are optional and can be used in any build of RTA-OSEK Component. In addition to the OSEK hook routines, RTA-OSEK defines two additional hooks:

- Stack Fault Hook.
- Overrun Hook.

RTA-OSEK hook routines are mandatory. The Stack Fault Hook is only used in the Extended build of RTA-OSEK Component and the Overrun Hook is only used in the Timing and Extended builds.

You will find out more about each of these hooks later in this chapter.

13.1 Enabling Hook Routines

In the RTA-OSEK GUI you can select the hooks that you want to use in your application. Have a look at Figure 13:1.

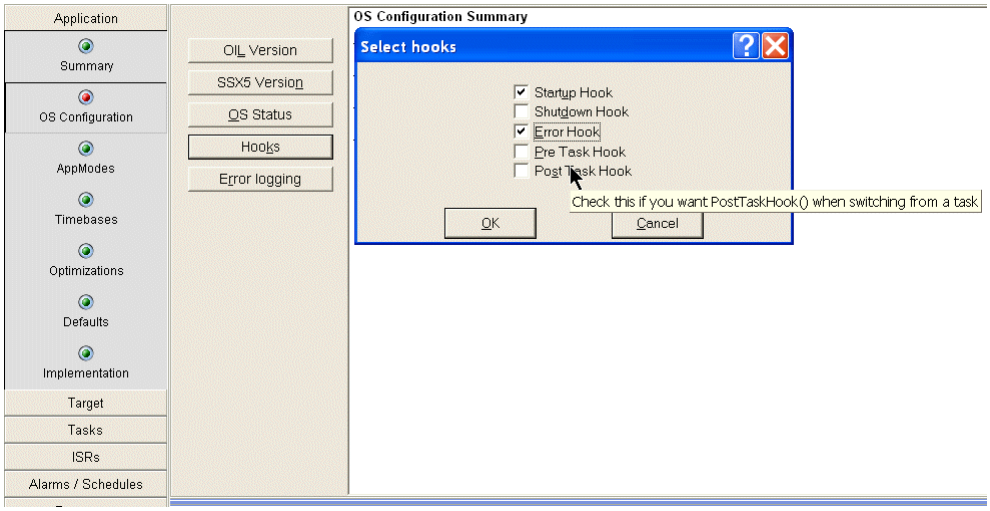


Figure 13:1 - Configuring OSEK Hooks for an Application

Figure 13:1 shows how the Startup Hook and Error Hook have been enabled (remember that OSEK hooks are optional).

Important: If you do not provide code for a hook that you have enabled, your program will not link correctly.

RTA-OSEK defines a set of macros that allow you to conditionally compile the optional hooks routines into your code. These macros are called:

```
OSEK_STARTUPHOOK
OSEK_SHUTDOWNHOOK
OSEK_PRETASKHOOK
OSEK_POSTTASKHOOK
OSEK_ERRORHOOK
```

These macros are only defined if the corresponding hook is enabled.

13.2 Mandatory Hooks

RTA-OSEK provides a Stack Fault and an Overrun Hook. These additional hooks are mandatory in the Timing and Extended builds.

If you use any extended tasks, you must provide a handling function for `StackFaultHook()` in your application code. You can find out more about the Stack Fault Hook in Section 13.7.

`OverrunHook()` is mandatory if you use the Timing or Extended builds of RTA-OSEK Component. You can use the predefined `OS_ET_MEASURE` macro to conditionally compile the Overrun Hook. Using this macro ensures that the code is not included in a Standard build of your application. The Overrun Hook is explained in Section 13.9.3.

13.3 Startup Hook

The Startup Hook is called by RTA-OSEK Component during the `StartOS(OSDEFAULTAPPMODE)` call after the kernel has been initialized, but before the scheduler is running. Figure 13:2 shows the execution of the Startup Hook relative to the initialization of RTA-OSEK Component.



Figure 13:2 - Execution of the Startup Hook

Code Example 13:1 shows how Startup Hook should appear in your code.

```
#ifdef OSEK_STARTUPHOOK

OS_HOOK(void) StartupHook(void) {
  /* Startup hook code. */
}

#endif
```

Code Example 13:1 - Using the Startup Hook

The Startup Hook is often used for the initialization of OSEK COM or initialization of target hardware (configuration and initialization of interrupts sources, for example).

13.4 Shutdown Hook

The **Shutdown Hook** is called during the execution of the `ShutdownOS()` API call. Figure 13:3 shows the execution of the Shutdown Hook with respect to a `ShutdownOS()` API call.



Figure 13:3 - Execution of the Shutdown Hook

Code Example 13:2 shows how Shutdown Hook should appear in your code.

```
#ifdef OSEK_SHUTDOWNHOOK

OS_HOOK(void) ShutdownHook(StatusType s) {
  /* Shutdown hook code. */
}

#endif
```

Code Example 13:2 - Using the Shutdown Hook

The Shutdown Hook is often used for shutting down COM.

You should not normally return from the Shutdown Hook. If you do, however, the behavior of RTA-OSEK Component is to enter an infinite loop running at OS level.

13.5 Pre and Post Task Hooks

The **PreTask Hook** is called by RTA-OSEK Component whenever a task moves into the *running* state. This means that the PreTask Hook will also be called whenever a task is resumed after preemption.

The **PostTask Hook** is called by RTA-OSEK Component whenever a task moves out of the *running* state. The PostTask Hook will be called when the task terminates and each time a task is preempted.

Figure 13:4 shows where the PreTask and PostTask Hooks are called relative to task preemption.

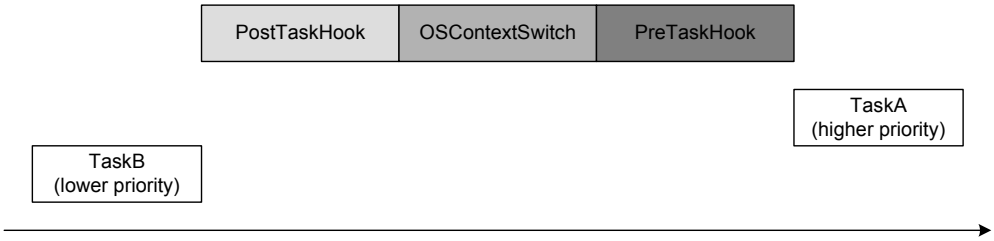


Figure 13:4 -The PreTaskHook() and PostTaskHook() Relative to Task Preemption

Code Example 13:3 shows how the hooks should appear in your code.

```
#ifdef OSEK_PRETASKHOOK

OS_HOOK(void) PreTaskHook(void) {
  /* PreTask hook code. */
}

#endif

#ifdef OSEK_POSTTASKHOOK

OS_HOOK(void) PostTaskHook(void) {
  /* PostTask hook code. */
}

#endif
```

Code Example 13:3 - The OSEK PreTaskHook and PostTaskHook

The PreTask and PostTask Hooks are called on entry and exit of tasks and for each preemption/resumption. This means that it is possible to use these hooks to log an execution trace of your application.

13.6 The Error Hook

All RTA-OSEK Component API calls return a status code. You can find out more about the status codes in the *RTA-OSEK Reference Guide*.

If the **Error Hook** is enabled, it is called from any API call that is about to return a status code that is not `E_OK`. The status code is passed into the Error Hook routine to determine the type of error.

Depending on the severity of the error you can decide whether to terminate (by calling `ShutdownOS()`) or to resume (by handling or logging the error and then returning from `ErrorHook()`).

Code Example 13:4 shows you the usual structure of the Error Hook.

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType status) {
  switch (status) {

    case E_OS_ACCESS:
      /* Handle error then return. */
      break;

    case E_OS_LIMIT:
      /* Terminate. */
      ShutdownOS(status);

    default:
      break;
  }
}

#endif
```

Code Example 13:4 - Suggested Structure of the Error Hook

13.7 The Stack Fault Hook

The Stack Fault Hook is called whenever RTA-OSEK Component detects a problem with stack management for extended tasks.

Portability: The Stack Fault Hook is only used in RTA-OSEK; it is not part of the OSEK OS standard.

`StackFaultHook()` is called from RTA-OSEK Component with 3 parameters:

- `StackID`.
  This will always be zero for targets with a single stack. Otherwise it will be an integer indicating which stack the fault applies to. The *RTA-OSEK Binding Manual* explains how stacks are numbered on your target.
- `StackError`.
  This is an integer indicating the cause of the error. `OS_EXTENDED_TASK_STARTING`:
  The task could not have the starting stack pointer set because the application stack pointer was already too high or too low. `OS_EXTENDED_TASK_RESUMING`:
  The task could not have the resuming stack pointer set because the application stack pointer was already too high or too low. `OS_EXTENDED_TASK_WAITING`:

The task could not be moved off the stack into the *waiting* state because it has used more stack than declared for it during configuration with the RTA-OSEK GUI.

- `Overflow`.
  This specifies the number of bytes on the stack beyond the number that the application was expecting.

The Stack Fault Hook is shown in Code Example 13:5.

```
OS_HOOK(void) StackFaultHook(
  SmallType StackID,
  SmallType StackError,
  UIntType  Overflow) {

  for (;;) {
    /* Loop forever. */
  }
}
```

Code Example 13:5 - The Stack Fault Hook

`StackFaultHook()` can only occur when the wrong stack usage information is entered into the RTA-OSEK GUI. Check the stack declarations for each task that has lower priority than the currently running task.

Important: You should not return from the `StackFaultHook()`. Entering the hook usually means that your stack is corrupt. If you do return from the hook then the behavior of your application is undefined.

Advanced Error Handling & Execution Monitoring Concepts

13.8 Advanced Error Logging

In Section 13.6 you learnt how errors could be handled using the Error Hook. The techniques you saw were adequate for coarse debugging. Sometimes, however, you will need to know more about the error.

You may wish to know, for example, which API call resulted in the error being generated and which parameters were passed to that API call. This information is available at run-time by configuring error logging using the RTA-OSEK GUI.

13.8.1 Configuring Advanced Error Logging

In RTA-OSEK, two levels of detail are available:

- Record the API name only.
- Record the API name and the associated parameters.

Figure 13:5 shows how the level of detail is defined in the RTA-OSEK GUI.

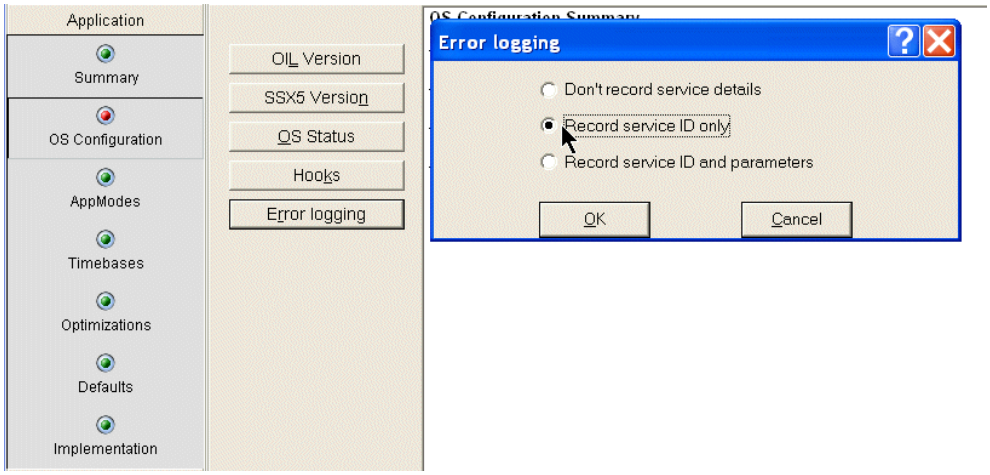


Figure 13:5 - Configuring Advanced Error Logging

If you choose not to record the service details, your application does not need to pay the additional overheads associated with collecting this information.

13.8.2 Using Advanced Error Logging

When error logging is enabled, RTA-OSEK provides a set of macros for accessing the name and the associated parameters of the API call that caused the error.

The API call causing the error is accessible using the `OSErrorGetServiceId()` macro. This macro returns an `OSServiceIdType` of the form `OSServiceId_<API name>`. If, for instance, an `ActivateTask()` call results in an error, `OSErrorGetServiceId` will return `OSServiceId_ActivateTask`.

The parameters to the API call are available using macros in the form shown in Code Example 13:6. A macro is defined for each parameter of each API call.

```
OSError_<API Name>_<API Parameter Name>
```

Code Example 13:6 - Advanced Error Logging

Using the `ActivateTask()` example again, `OSError_ActivateTask_TaskId` will return the `TaskId` parameter passed to `ActivateTask()`. This additional error logging information can be usefully incorporated into the `ErrorHook()` code. This is shown in Code Example 13:7.

```
#ifdef OSEK_ERRORHOOK

OS_HOOK(void) ErrorHook(StatusType status) {

  OSServiceIdType callee;

  switch (status) {

    case E_OS_ID:
      /* API call called with invalid handle. */
      callee = OSErrorGetServiceId();
```

```
      switch (callee) {

        case OSServiceId_ActivateTask:
          /* Handle error. */
          break;

        case OSServiceId_ChainTask:
          /* Handle error. */
          break;

        case OSServiceId_SetRelAlarm:
          /* Handle error. */
          break;

        default:
          break;
      }

      break;

    case E_OS_LIMIT:
      /* Terminate. */
      ShutdownOS();

    default:
      break;
  }
}

#endif
```

Code Example 13:7 - Additional Error Logging Information

The macros for obtaining the API name and the associated parameters should only be used from within the Error Hook. The values they represent do not persist outside the scope of the hook.

Important: When you use extended error logging the value returned by `OSErrorGetServiceId()` may be misleading. This generally happens when API calls have a side effect. For example if you send a message using COM, a possible side effect is to activate a task. If that task activation results in an error then `OSErrorGetServiceId()` will return `OSServiceId_ActivateTask` even though the API call that you made was `SendMessage()`.

13.9 Measuring and Monitoring Execution Time

RTA-OSEK Component provides facilities for measuring the execution times of user code at the kernel level. These facilities must be used in conjunction with the add-on Execution Time Correction (ETC) program.

Important: Before measuring code timings, you must run the ETC program. Otherwise, the values you obtain will be incorrect.

Normally you will use the Timing build to measure the execution time for your application. However, the timing measurement facilities are available in both the Timing and Extended build of RTA-OSEK Component. If you use timing measurement facilities in Extended build, the times that you obtain will include the additional overhead required to perform more extensive error checking.

The kernel and application code are identical for Timing build and Standard build, other than the code needed to support the timing measurement.

13.9.1 Enabling Timing Measurement

For timing measurement a 'stopwatch' source must be provided. This is usually a free running counter on your target hardware. RTA-OSEK Component accesses the stopwatch using the `GetStopwatch()` callback function. This function is shown in Code Example 13:8.

```
OS_NONREENTRANT(StopwatchTickType)
GetStopwatch(void) {
  return CurrentValueOfFreeRunningCounter;
}
```

Code Example 13:8 - Accessing the Stopwatch

If the stopwatch runs slower than the processor clock, subtraction of two values to provide an execution time has inherent uncertainty. As a result of this you must also provide a function that allows RTA-OSEK Component to compensate for this uncertainty. Code Example 13:9 shows how `GetStopwatchUncertainty()` is used.

```
OS_NONREENTRANT(StopwatchTickType)
GetStopwatchUncertainty(void){
  return Uncertainty;
}
```

Code Example 13:9 - Compensating for Uncertainty

The returned uncertainty value is usually 0 if the stopwatch tick length is the same as a CPU instruction cycle and 1 otherwise.

You may find that there are some systems where the uncertainty can be greater than 1. This is rare, but you can declare that the stopwatch runs at 40MHz and the counter hardware only runs at 10MHz. You can then multiply the counter value by 4 in `GetStopwatch()` and report an uncertainty of 4.

Important: Implementations of `GetStopwatchUncertainty()` and `GetStopwatch()` must be provided if you are using the Timing or Extended builds. If you do not provide these functions, your program will not link correctly.

13.9.2 Measuring Execution Times

When your application uses the Timing or Extended builds, RTA-OSEK Component measures the execution times of each task and Category 2 ISR in your application.

RTA-OSEK Component maintains a log of the longest observed execution time over all executions for each task or Category 2 ISR. You can get the largest observed execution time for each task and ISR using the `GetLargestExecutionTime()` API call.

13.9.3 Setting Timing Budgets

The execution time budgets for each task and Category 2 ISR can be set in your application. These values are optional and do not have to be supplied. An execution budget has been specified in Figure 13:6.

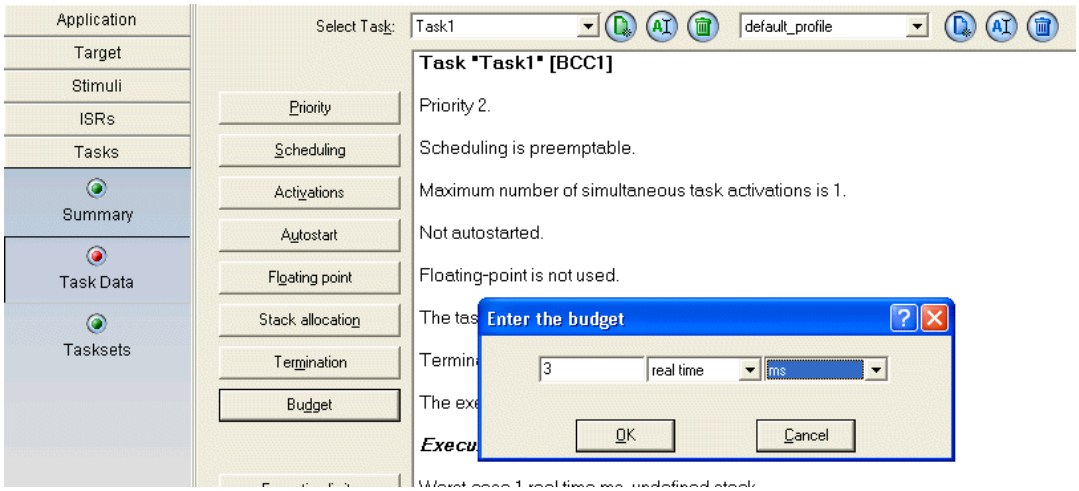


Figure 13:6 - Specifying the Execution Time Budgets

When using the Timing or Extended build, RTA-OSEK Component will check to see whether tasks or Category 2 ISRs consume more time than is specified in the budget. If the budget is exceeded, RTA-OSEK Component will call the `OverrunHook()` when the task terminates (or, in the case of an extended task, when it calls `WaitEvent()`). This allows you to log the budget overrun.

The prototype for `OverrunHook()` is shown in Code Example 13:10.

```
#ifdef OS_ET_MEASURE

OS_HOOK(void) OverrunHook(void) {
  /* Log budget overruns. */
}

#endif
```

Code Example 13:10 - The OverrunHook Prototype

You should be aware that, for extended tasks, the execution time is reset to zero at the start of the task and when resuming from `WaitEvent()`. Normally the budget is used to check the execution time between consecutive `WaitEvent()` calls.

You should also be aware that the execution time is only sampled by RTA-OSEK Component when a task is preempted by another task or ISR.

In some unusual circumstances, it is possible for a budget overrun to be missed. This could happen when the interval between preemptions approaches the maximum interval that can be measured by a `StopwatchTickType`. The range of a `StopwatchTickType` is target dependent, but is normally 2^{16} or 2^{32}.

13.9.4 Obtaining Blocking Times

You can prevent timing analysis from being too pessimistic. To do this you will need to provide accurate timings both for the time spent inside critical sections protected by resources and for the amount of time that interrupts are disabled.

The timing API call `GetStopwatch()` or `GetExecutionTime()` can be used to get the current stopwatch value immediately before and immediately after these sections of code. Additional code must be provided to hold intermediate values and to maintain the 'high watermark' times.

Any code that your application uses to obtain execution times should be conditionally compiled. RTA-OSEK provides the macro `OS_ET_MEASURE`, which allows you to do this. Code Example 13:11 shows an example of conditional compilation when getting the time that a resource is held.

```
TASK(Task1) {
  ...
#if defined(OS_ET_MEASURE)

  /* Get time for GetExecutionTime() call */
  /* itself. */
  start = GetExecutionTime();
  finish = GetExecutionTime();
  correction = finish - start -
               GetStopwatchUncertainty();

  /* Measure resource lock time. */
  start = GetExecutionTime;
```

```
#endif

  GetResource(Resource1);
  /* Critical section. */
  ReleaseResource(Resource1);

#if defined(OS_ET_MEASURE)

  finish = GetExecutionTime();
  /* Calculate amount of time used. */
  used = finish - start - correction +
           GetStopwatchUncertainty();
#endif
}
```

Code Example 13:11 - Use of Conditional Compilation

13.10 Measuring and Monitoring Stack Use

Each task profile can include an optional stack space figure that is used by RTA-OSEK to calculate the worst-case stack usage of your entire application.

The figures that you supply should represent the worst-case stack used by each task and should be the sum of the space required by the task, plus the space required for each function call made by the task on the worst-case path in the function call hierarchy.

Normally you would obtain this information from your linker or from your debugging/emulation environment. This is the preferred method, however, if your toolchain does not provide this, you can use internal facilities provided by RTA-OSEK Component to measure these figures.

The `GetStackOffset()` API call is used for stack measurement in RTA-OSEK Component. On targets that have a single stack, `GetStackOffset()` returns a scalar value indicating the number of bytes of stack space consumed. If your target has multiple stacks, however, `GetStackOffset()` returns a data structure containing the number of bytes used on each stack. The *RTA-OSEK Binding Manual* for your target will tell you how to extract stack space information from this data structure.

The values returned are measured from the initial location of the stack pointer. So, when you make the call in a task or ISR, the value returned will include the stack consumed by C startup code, the main program (the idle task) and all pre-empted tasks or ISRs (including the space consumed by OS context idle task and main program). The figures returned by `GetStackOffset()` do not include the stack space required for the call itself. Figure 13:7 shows the size returned by `GetStackOffset()` when it is called from task `Task3`.

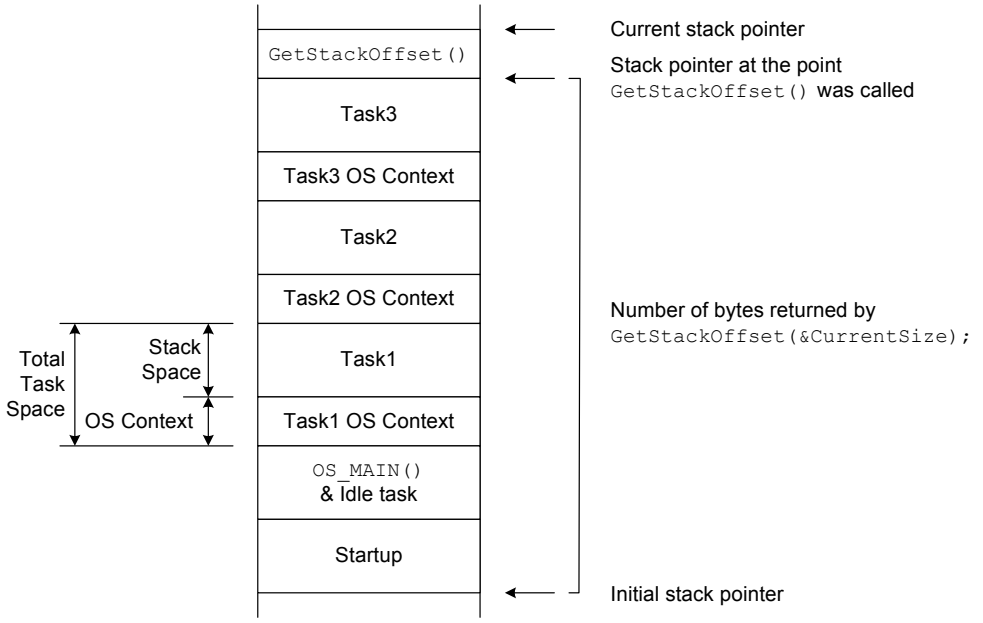


Figure 13:7 - Stack Diagram

To calculate the worst-case stack usage for each task or ISR, you will need to make a `GetStackOffset()` call at each leaf of your function call hierarchy. You will also have to calculate the maximum value returned by these calls.

If you have leaves that are library functions then you will need to make a `GetStackOffset()` call in the parent function and determine the worst-case stack space of the library call. You can find the worst-case stack space requirements for the RTA-OSEK Component API in the *RTA-OSEK Binding Manual* for your target.

If you make calls to other libraries at the leaves of your call hierarchy, you must contact the vendor to obtain the worst-case stack requirements for the library calls you make.

Code Example 13:12 shows a task that makes a number of function calls. It shows the placement of `GetStackOffset()` calls required to measure stack usage.

```
StackOffsetRefType Measurement1;
StackOffsetRefType Measurement2;
StackOffsetRefType Measurement3;

void f1(void) {
  ...
  GetStackOffset(&Measurement1);
  ActivateTask(TaskB);
  ...
}
void f2(void) {
  ...
  f3();
  GetStackOffset(&Measurement2);
  memcpy(x,y);
  ...
}
```

```
void f3(void) {
  ...
  GetStackOffset(&Measurement3);
  ...
}
TASK(Task3) {
  f1();
  f2();
  TerminateTask();
}
```

Code Example 13:12 - Measuring Stack Usage

The worst-case stack usage for Code Example 13:12 is the maximum value of:

1. `Measurement1` + Stack space requirements for `ActivateTask()` call
   – Stack offset for pre-empted task
   – Stack space for OS context
2. `Measurement2` + Stack space requirements for C library `memcpy` call
   – Stack offset for pre-empted task
   – Stack space for OS context
3. `Measurement3` – Stack offset for pre-empted task
   – Stack space for OS context

The easiest way to measure the stack space required per task (without having to worry about the size of the stack at the point of pre-emption) is to run each task in isolation with interrupts disabled.

Normally you would make a `GetStackOffset()` call immediately after `StartOS()` to baseline the stack pointer. You can then use this in your calculation.

This method, however, will only work correctly if `StartOS()` returns. If you have autostarted tasks that never return, you will never return from `StartOS()` and the baseline value will never be set.

If this happens, you must baseline your stack in some other way. You could do this, for example by recording the value of the stack pointer prior to making the `StartOS()` call.

13.11 Catching Errors at Compile Time

So far you have learnt about error checking at run-time. It is better, however, to try and remove errors at compile time. RTA-OSEK can help with this if you use the static interface wherever possible.

When you use the static interface you can only use API calls and objects that are valid for a specific task or ISR. Consequently you can only make API calls that RTA-OSEK has already checked for validity during the build process.

Any attempt to use invalid parameters will be detected by the compiler, which can potentially save you a significant amount of debugging time.

13.12 Run-time Fault Tolerance

The status code returned by an API call can be checked at run-time. This means that you can build some degree of run-time fault tolerance into your application.

This may be useful if you choose not to provide an Error Hook, but need to check for error conditions that can occur in the Standard build (such as, `ActivateTask()` returning `E_OS_LIMIT`). Code Example 13:13 shows you how this can be done.

```
if (ActivateTask(Task1) != E_OK) {

  /* Handle error during task activation. */

}
```

Code Example 13:13 - "On-the-Fly" Error Checking

13.13 Imprecise Computation

Because the overheads imposed by the additional timing facilities are small, the Timing build can be used for production code. You can exploit this fact to perform imprecise computation.

Imprecise computation is useful in applications that iteratively converge on a result. If a task has not traveled down the worst-case path, then it will not have run in the worst-case execution time. If this is the case, any 'spare' CPU cycles available to the task can be used to refine a result. This technique is illustrated in Code Example 13:14.

```
TASK(Task1) {
  ...
  while ((Budget - GetExecutionTime()) >
           MaxLoopTime) {

    /* Perform iterative refinement of output. */
  }
}
```

Code Example 13:14 - Imprecise Computation

Error Handling and Execution Monitoring Summary

- OSEK provides facilities for debugging through its hook mechanisms.
- The Startup, Shutdown, PreTask and PostTask Hooks allow you to profile your application at run-time.
- The Error Hook provides a mechanism for trapping exceptional conditions at run-time. It can provide a resumption model of exception handling.
- Further information on the source of an error is available through macros accessible in the Error Hook.

- RTA-OSEK provides additional support for execution time and stack usage monitoring. The Timing and Extended builds of RTA-OSEK Component allow you to measure the execution times of user provided code

14 Modeling the Application for the RTA-OSEK Planner

You have seen how systems receive stimuli and generate responses. You have also seen how stimulus/response models are used in RTA-OSEK to capture a specification of system behavior and how you can use that specification in the design process.

If you are building a real-time system, there is an extra dimension to this model. A specification of the required real-time performance is needed.

The meaning of the term 'real-time' is often misunderstood. People tend to think that real-time means 'real-fast'. Real-time systems are systems where every response must always be generated on time whenever the associated stimulus arrives.

You might need to generate a response in minutes, hours or even years after the stimulus arrives. So, no matter how long it takes, if the response must be generated within a specific time frame, then the system is real-time.

The latest time by which a response must be generated is called its deadline. Figure 14:1 illustrates the relationship between stimuli, responses, deadlines and periods.

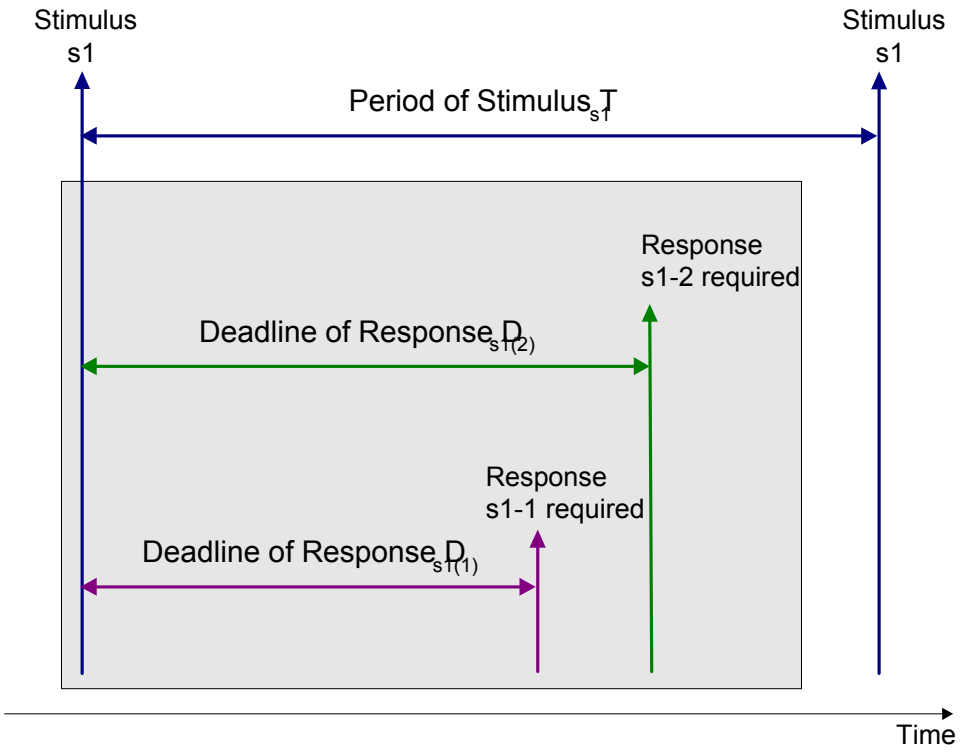


Figure 14:1 - A Stimulus/Response Model

A deadline will be met if the response implementation generates the response before the deadline. Figure 14:2 shows a task being used as a response implementation. Here the task generates the response at the end of the execution.

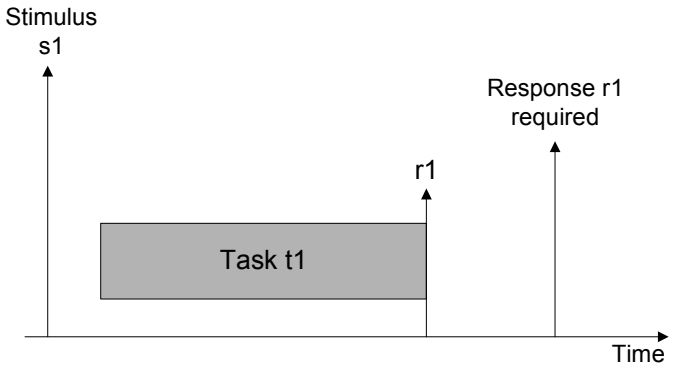


Figure 14:2 - A Task used as a Response Implementation

The time taken to generate the response is called the **response time**. If the response is generated before the deadline, then the deadline is met.

The response need not be generated at the end of the task and more than one response can be generated from a response implementation. Figure 14:3 shows a task response being generated before the end of the task.

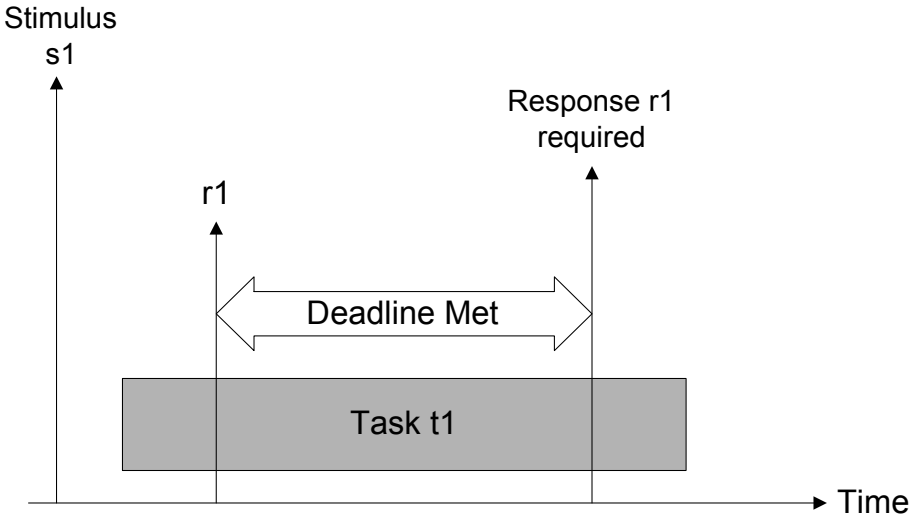


Figure 14:3 - Generating a Response before the Deadline Expires

In a real-time system, you need to show that every response occurs before its associated deadline. The RTA-OSEK GUI uses RTA-OSEK Planner to calculate the worst-case response time for each response in your application. It then checks that all the responses meet their deadlines.

For each task, the worst-case response time for a response implementation consists of:

- Worst-case execution time.
  This is the longest time between the task starting and the task terminating, assuming that there are no preemptions.
- Interference.
  This is the maximum time that the task is preempted by other higher priority tasks and ISRs in the system.
- Blocking.
  This is the maximum time that the task is prevented from running by lower priority tasks.

RTA-OSEK Planner calculates the interference suffered by each task or ISR. It calculates this using the worst-case execution time for the response implementation and the times that resources are used and interrupts are disabled (the blocking times).

So, RTA-OSEK Planner can calculate the worst-case response time with the worst-case execution time and blocking times for the response.

To analyze a real-time system you must provide:

- A description of the software architecture in terms of the tasks, interrupts, tasksets, resources, counters, alarms and schedules.
- A stimulus/response model defining the timing relationship between executable objects. This defines the periods and deadlines for your application.
- The execution times for each task and ISR.
- Target specific timing information.

You learnt about describing the software architecture in previous chapters. You will now learn about defining the stimulus/response model, execution times and target specific timing information.

When you provide data to model the system for analysis, there are two important principles that need to be followed:

- Accuracy.
  It is important to provide data that is as accurate as possible.
- Pessimism.
  If you cannot guarantee that your data is accurate, you must supply data that is pessimistic. You can make your data pessimistic by supplying execution times that are longer than the actual execution times, for example. You could also declare delays between stimuli as shorter than the actual delays.

Important: When you provide data for analysis, be careful not to underestimate execution times and to overestimate minimum periods. If you do this then RTA-OSEK could say that your system is schedulable when it is, in fact, unschedulable.

14.1 Configuring Applications for Analysis

If you want to build an application for timing analysis you will need to follow these rules:

- Upward activation of tasks is not allowed. A task can only activate tasks of lower priority.
- Tasks must be assigned unique priorities.
- The `Schedule()` API call cannot be used to force rescheduling to take place.

The RTA-OSEK GUI allows you to enforce these rules, using the Application Optimizations. Figure 14:4 shows where these settings can be found.

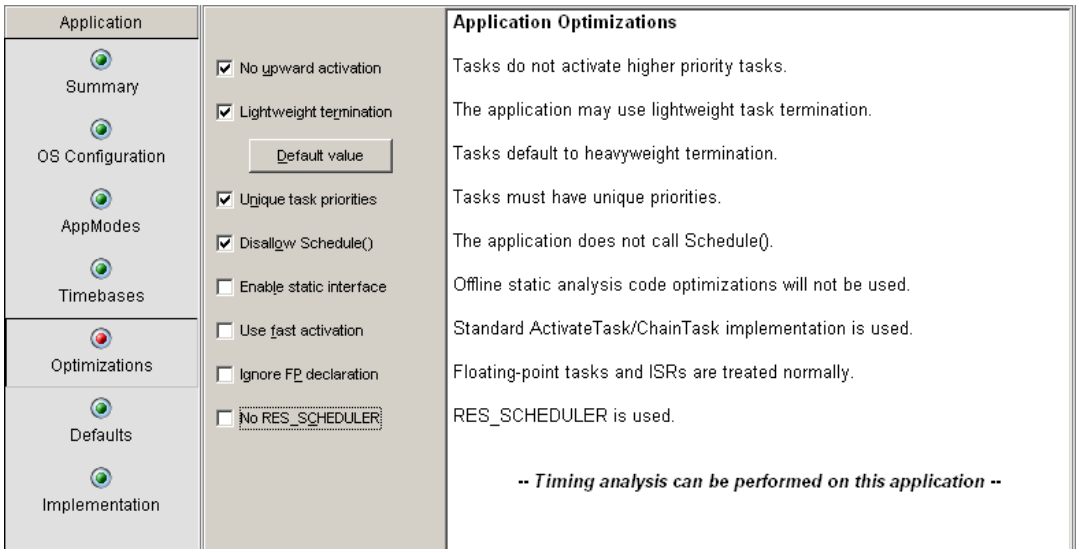


Figure 14:4 - Configuring the Application Optimization Settings

When you try to analyze a system, RTA-OSEK Planner will tell you if an application is not suitable for analysis.

14.2 Defining Stimulus/Response Timing Relationships

You have already seen how you can create stimulus/response models to build applications using the RTA-OSEK GUI. You'll now see some of this information again, but this time you'll also see how you can add extra information for timing analysis.

14.2.1 Stimulus Arrival Types and Patterns Revisited

Remember that each stimulus is associated with an arrival type. The arrival type specifies the class of the stimulus.

You saw that there are 3 arrival types:

- Bursty.
- Periodic.

- Planned.

Bursty stimuli are used to model simple cases where a stimulus is captured by an interrupt directly. Periodic and planned stimuli are used to model more complex arrivals, where the stimulus is modeled by an alarm attached to a counter or by arrivalpoints on a schedule.

Each type of arrival has a distinct arrival pattern. Let's look at each of these arrival patterns in more detail.

14.2.2 Bursty Arrival Patterns

Bursty arrival patterns allow you to define a set of rules called **bursting clauses**. These clauses describe the arrival pattern of the stimulus.

A simple bursty arrival pattern could specify the arrival of a periodic timer interrupt. In Figure 14:5 you can see that a bursty arrival for a 10ms periodic interrupt has been defined. In this case, a single bursting clause is used.

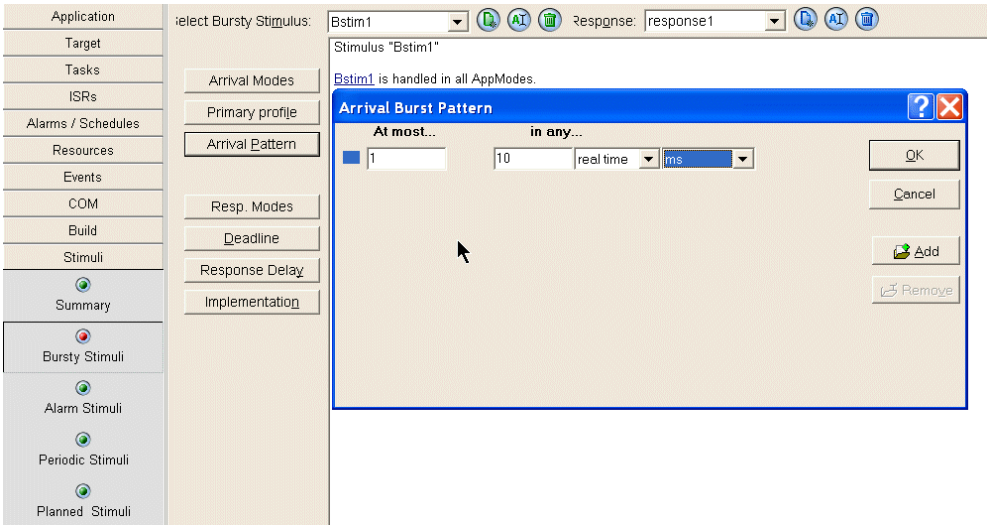


Figure 14:5 - Single Bursting Clauses

Figure 14:6 shows a more complex example of a bursty arrival pattern using multiple arrival rules.

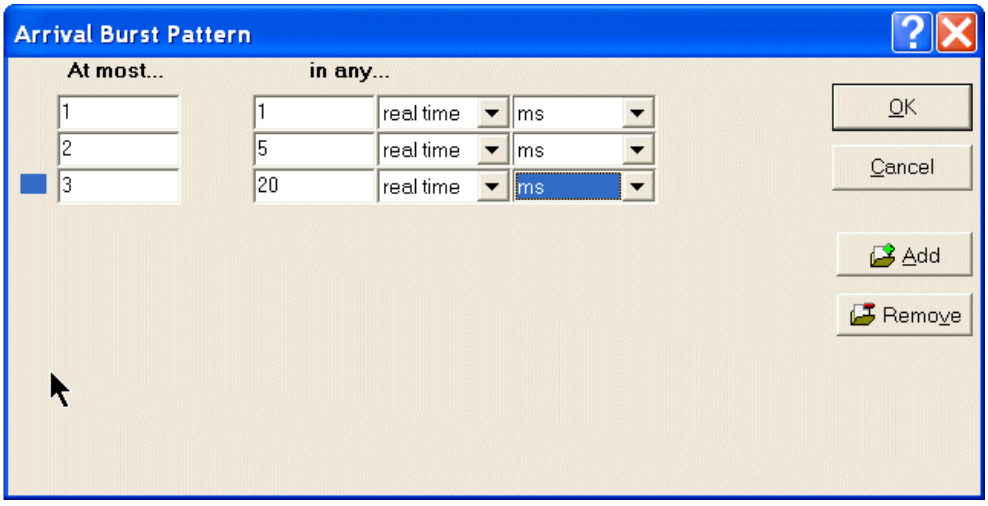


Figure 14:6 - Multiple Bursting Clauses

In Figure 14:6, the bursting clause of the transaction specifies the following rules. The stimulus will occur:

Rule 1. No more than once in any one millisecond.

Rule 2. No more than twice in any five milliseconds.

Rule 3. No more than three times in any twenty milliseconds.

You can combine these rules to form a worst-case arrival pattern as follows:

- 0ms, 1ms, 2ms, 3ms ... (Rule 1 allows the minimum inter-arrival time of 1ms).

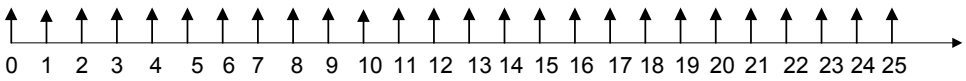


Figure 14:7 - Rule 1 Arrival Pattern

- 0ms, 1ms, 5ms, 6ms, 10ms, 11ms ... (Rule 2 prevents more than 2 arrivals in a period of 5ms, so bursts of 2 are separated by 5ms periods).

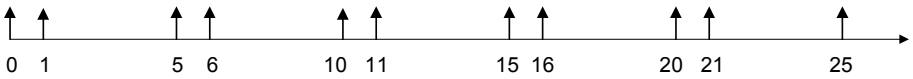


Figure 14:8 - Rule 2 Arrival Pattern

- 0ms, 1ms, 5ms, 20ms, 21ms, 25ms ... (Rule 3 prevents more than 3 arrivals within a 20 millisecond interval).



Figure 14:9 - Rule 3 Arrival Pattern

If more than one arrival rule is given, another rule covers the values that are allowed. If values are arranged in increasing order, each successive pair of values (arrivals, interval) must be greater than the previous pair. The rate of arrivals (that is, arrivals/interval) must strictly decrease.

Following on from the previous example you can see that:

- 1 time < 2 times < 3 times
- 1ms < 5ms < 20ms
- 1/ms (1 time in 1ms) > 0.4/ms (2 times in 5ms) > 0.15/ms (3 times in 20ms)

Generally, pessimism in analysis will become lower the more bursting clauses that are given. However, if the bursting interval is greater than the longest busy period for the system, the arrival rule doesn't give you any benefit. So, in this example, if you know that the system will never run for longer than 20ms before the idle task runs, then Rule 3 will not improve the accuracy of the analysis.

The idle task is the lowest priority task in the system and will only run when all other tasks and ISRs are in the *suspended* or *waiting* state.

The number of arrivals allowed during an operating cycle of a system can be limited to a finite number. The operating cycle is the time between system start and the point at which it is reset. In this case the 'forever' interval can be used to limit the number of arrivals.

A bursting clause of '1 times in forever' means that the arrival of the event can only occur once during the operating cycle of the system. This could be used to represent the triggering of a one-off safety device, such as an airbag in a vehicle. Have a look at Figure 14:10 to see how the clause has been specified.

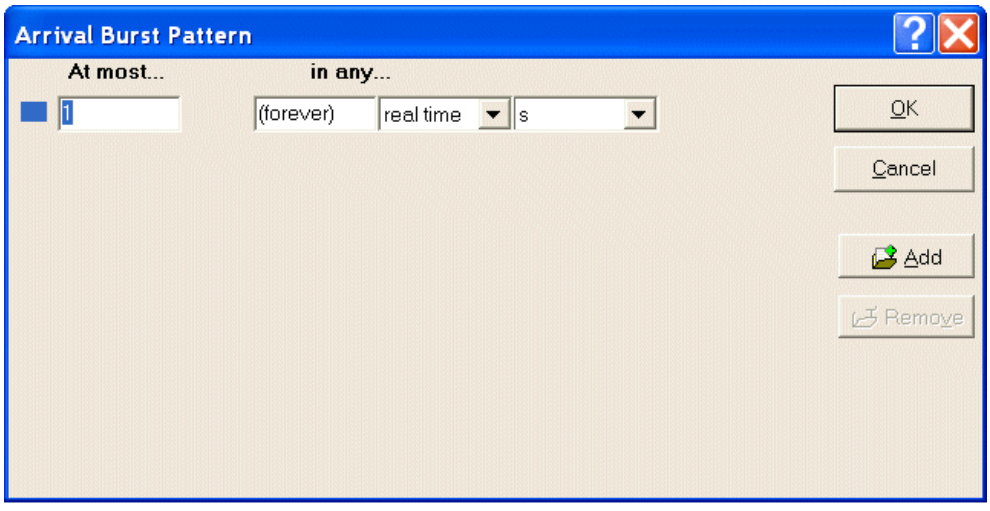


Figure 14:10 - Specifying a 'One Time in Forever' Bursting

14.2.3 Periodic Arrival Patterns

Periodic arrival patterns specify how often a stimulus arrives. This information is required for generating run-time information, such as alarm or schedule periods, and is also used to provide the implicit deadlines for analysis.

14.2.4 Planned Arrival Patterns

Planned arrival patterns are not specified at the stimulus/response modeling stage. The arrivals are planned at design time during the construction of the plan. RTA-OSEK Planner uses the timings on the plan. It then calculates the relative periods of stimuli and implicit deadlines of associated responses.

14.2.5 Setting Deadlines for Responses

Responses can have implicit deadlines. A 20ms periodic stimulus, for instance, may have to generate its response once in 20ms.

Responses can also have explicit deadlines. For example a 20ms stimulus might have to generate its response no later then 10ms after arrival.

Figure 14:11 shows an example of an explicit response deadline.

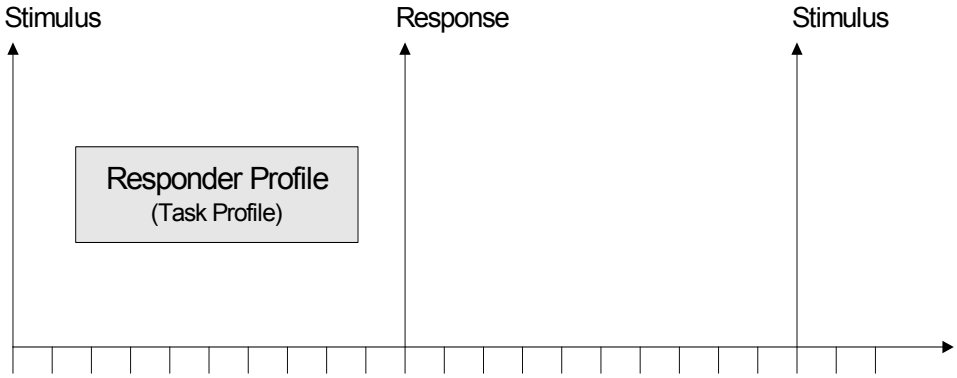


Figure 14:11 - Explicit Response Deadline

All responses have an implicit deadline, depending on whether the response is generated by a task or by an ISR and the properties of that executable object.

Tasks must complete before their next activation or, in the case of tasks with queued activation, before the queue is filled. ISRs must also complete before they are next triggered unless you have specified that the interrupts are buffered.

Response deadlines can be specified to provide additional timing performance constraints. The deadline is the elapsed time after the occurrence of the stimulus by which the response must be generated. Figure 14:12 shows you how explicit deadlines are defined using the RTA-OSEK GUI.

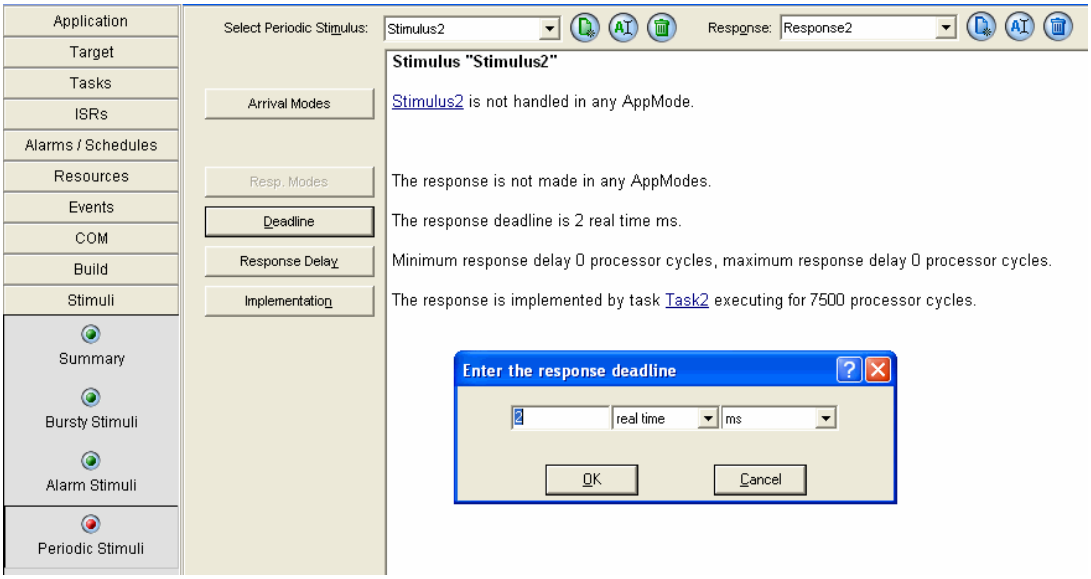


Figure 14:12 - Specifying a Response Deadline

14.2.6 Specifying Response Generation Time

The implementation of a response is performed by a task or ISR. The RTA-OSEK GUI assumes, by default, that the response will have been generated when the task or ISR terminates.

This can result, however, in pessimistic schedulability analysis. Let's look at an example where a stimulus occurs every 10ms and the associated response must be generated 1ms later.

If the response is generated by a task that executes for 2ms, then this system will not be schedulable. It isn't possible for the task to complete before the deadline. However, if you know that the response is generated after the task has been running for 0.5ms, the response generation time can be set to 0.5ms after the task start. The deadline can now be met.

So, in this example, you can see that there is an implicit and an explicit deadline on the task execution.

There is a 1ms explicit deadline from the arrival of the stimulus and a 10ms implicit deadline from the task period for the task to complete execution.

For each response implementation you can specify how much execution time must elapse before the response is generated. Have a look at Figure 14:13.

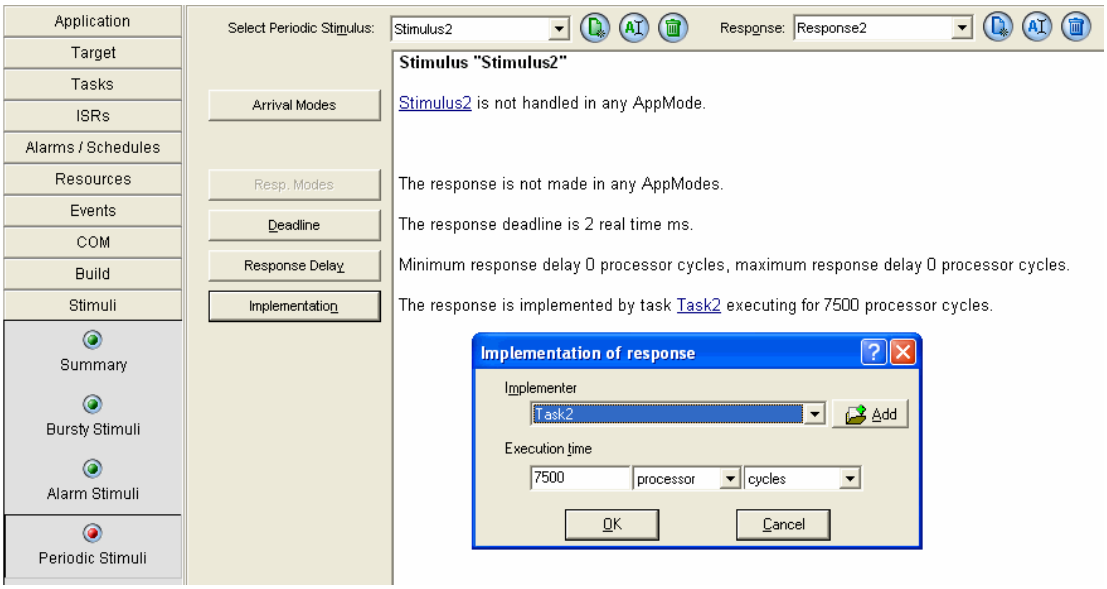


Figure 14:13 - Specifying Execution Time to Elapse before Response Generation

14.3 Capturing Execution Information

If you want to perform timing analysis on your application, the execution time behavior of each task and ISR in the system must be known.

You can determine this statically by counting CPU cycles or by using static timing analysis tools. Another way to do this is to use the Timing build of RTA-OSEK Component to measure execution times. You will find out more about the Timing build later in this guide.

If you supply the worst-case stack usage for each task and ISR, the RTA-OSEK GUI can provide facilities for calculating the worst-case stack usage. The worst-case stack usage information for each task and ISR is often available from your compiler.

The execution characteristics of tasks and ISRs are declared in **execution profiles**. For analysis you must define at least one execution profile for each task and ISR. You can also use multiple profiles, which are explained in Section 14.6.

The execution profile declares the worst-case execution time and worst-case stack usage of the corresponding task and ISR. Worst-case execution times are usually determined by the amount of code executed, so they are measured in processor cycles. This means that if you change the CPU clock rate, the execution time for your tasks and ISRs will scale automatically.

Tasks that perform imprecise computation are an exception to this rule. This type of task executes until it observes a value in a particular time. You should, therefore, express execution time using 'real-world' time units. Worst-case stack usage figures are specified in bytes.

14.3.1 Primary and Activated Profiles

Each task and ISR in your application will be associated with at least one **profile**. A profile is used to:

- Capture execution (timing and stack usage) information about the task or ISR.
- Indicate whether the task or ISR can be used in the capture or generation of a stimulus. If it can be used to capture a stimulus directly, to drive a counter or to drive a schedule, then it is a primary profile otherwise it is an activated profile.

RTA-OSEK assumes, by default, that all ISRs are primary profiles and that all tasks are activated profiles.

You will only need to change this setting in a few cases. When you need to drive a counter or schedule using a task rather than an ISR, for instance, the task will need to be a primary profile.

For tasks or ISRs with a single profile, the profile is accessed using the task or ISR name. For multiple profiles (which you can find out more about in Section 14.6) they are accessed using dot notation. For example, if `Task1` has profiles `Profile1` and `Profile2`, these are accessed using the names `Task1.Profile1` and `Task1.Profile2` respectively.

There are restrictions on how profiles can be used by your application:

- Single primary profile.
  Can be associated with exactly one bursty stimulus, exactly one advanced schedule or one or more counters and ticked schedules. For the first two of these options you cannot expect that two bursts or advanced activations will line up in time. For the last option, however, you know that the profile is being ticked at a constant rate. This means that it is feasible to tick more than one counter/schedule (even if the tick rates are not identical).
- Multiple primary profiles
  Can be associated with different stimuli as long as they are buffered by and execution profile.
- Single activated profile.
  Can be associated with exactly one primary profile.
- Multiple activated profile.
  Can either be associated with exactly one counter/schedule or each profile must be driven by a different primary profile.

14.3.2 Tasks and ISRs

The worst-case execution time for tasks and ISRs is measured from the start of the first machine code instruction of the task entry function, through the longest path in time and then to the end of the 'return' instruction. It excludes the effects of preemption or interrupts.

The worst-case execution times for Category 1 ISRs must make sure that the effects of any cache or instruction pipelines are at their most pessimistic.

The worst-case stack usage for tasks and Category 2 ISRs is taken from the entry function. It must also include the worst-case nested function call sequence, but the stack cost of entry to the task or Category 2 ISR does not need to be accounted for. This is added automatically during analysis.

Worst-case stack usage for Category 1 interrupt handlers must include the processor interrupt stack frame as well as the stack consumed by the handler.

Figure 14:14 shows how the worst-case execution time and stack usage are entered into the RTA-OSEK GUI.

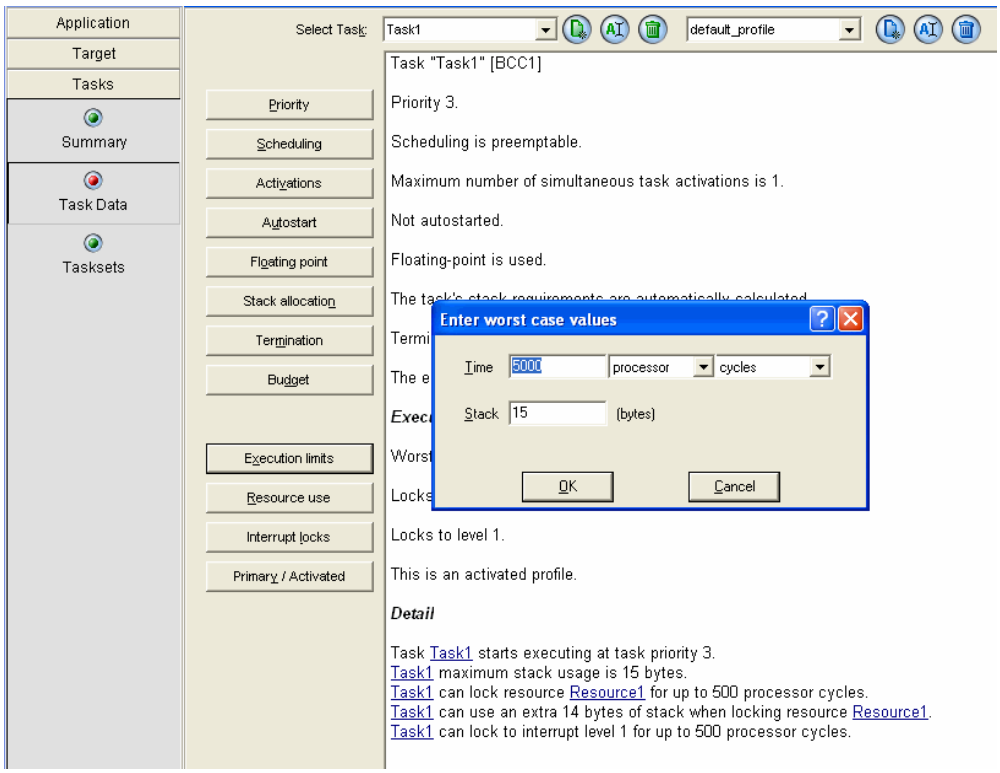


Figure 14:14 - Specifying the Worst-Case Values

In this example, the task uses 150 processor cycles and consumes 50 bytes of stack space in the worst-case.

14.3.3 Modeling the Idle Task

If the idle task makes any RTA-OSEK Component API calls other than the initial call to `StartOS(OSDEFAULTAPPMODE)`, it can introduce blocking. This must be considered in the analysis.

A profile for the idle task is specified in the same way as for any other task. If the idle task has no deadlines to meet, however, the exact value of the execution time specified is irrelevant.

You should be aware that the worst-case stack usage for the idle task is measured from the initial stack pointer value, normally set in the C run-time startup code.

14.3.4 Resource and Interrupt Locks

Tasks and ISRs that get resources or disable interrupts can block the execution of higher priority tasks and ISRs. Let's look at an example of a system that contains two tasks. The tasks are called `Task1` and `Task2` and they share a resource. `Task2` has a higher priority than `Task1`.

If `Task2` becomes *ready* when `Task1` owns the resource, it is **blocked** until `Task1` releases the resource. Have a look at the illustration in Figure 14:15.

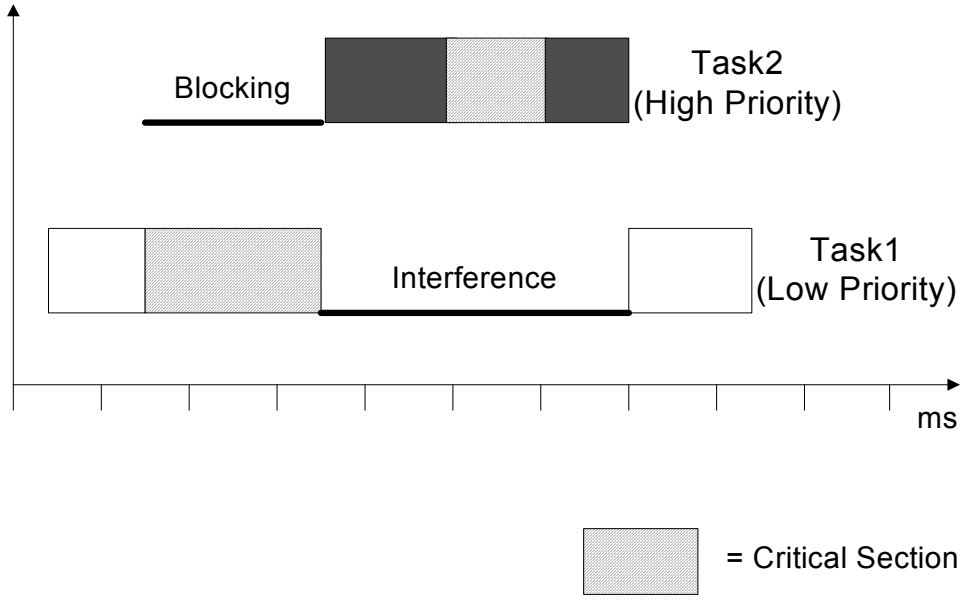


Figure 14:15 - Task Blocking and Interference

To determine whether your application is schedulable, RTA-OSEK Planner must know how long resources are held and how long interrupts are disabled for.

During analysis, it is assumed that resources are held at a time that gives the worst-case response time. This means that RTA-OSEK Planner does not need to know the time when the resource is held relative to the start of the task or ISR that gets the resource.

Locking times are specified in the RTA-OSEK GUI using the **resource use** and **interrupt lock** sections of the execution profile. Locking times, like execution times, are usually specified in processor cycles.

You can reduce the pessimism in the analysis by supplying accurate timing values. If you do not specify resource and interrupts locking times, then RTA-OSEK Planner assumes that the resource is held or that the interrupt is disabled for the entire execution time of the task or ISR.

Figure 14:16 shows how resource use times are specified in the RTA-OSEK GUI. Only the longest single execution time needs to be specified.

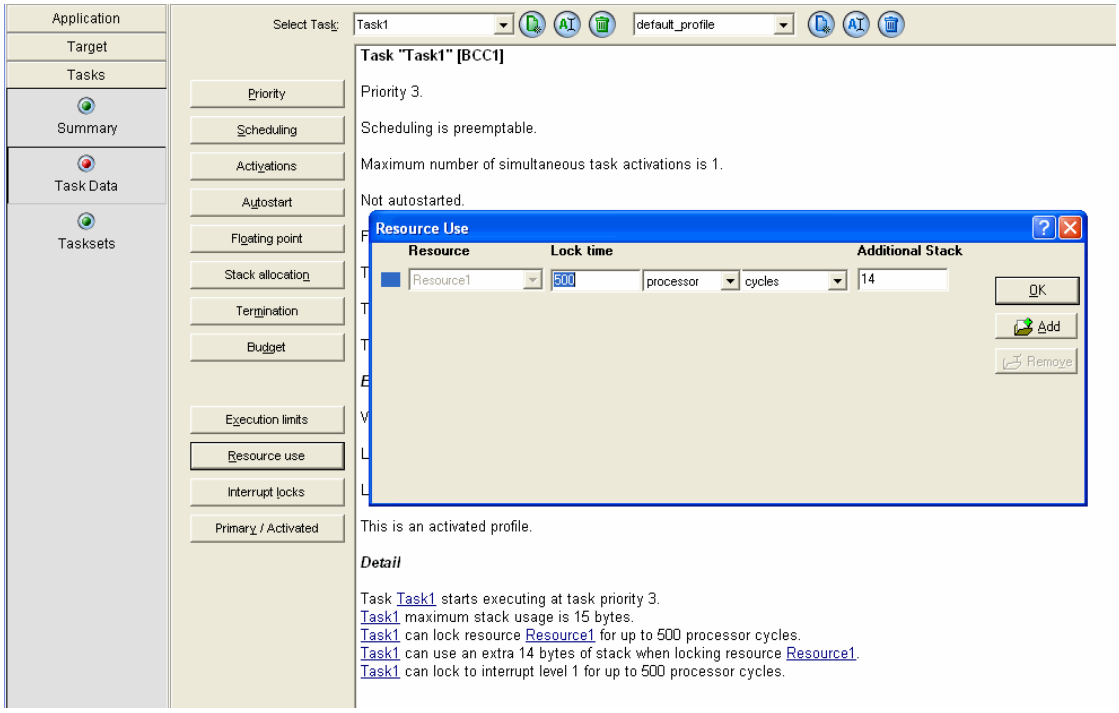


Figure 14:16 - Specifying Resource Use Times

Figure 14:17 shows how interrupt lock times are specified in the RTA-OSEK GUI. Again, for interrupt locking times, only the longest single execution time for the lock needs to be specified.

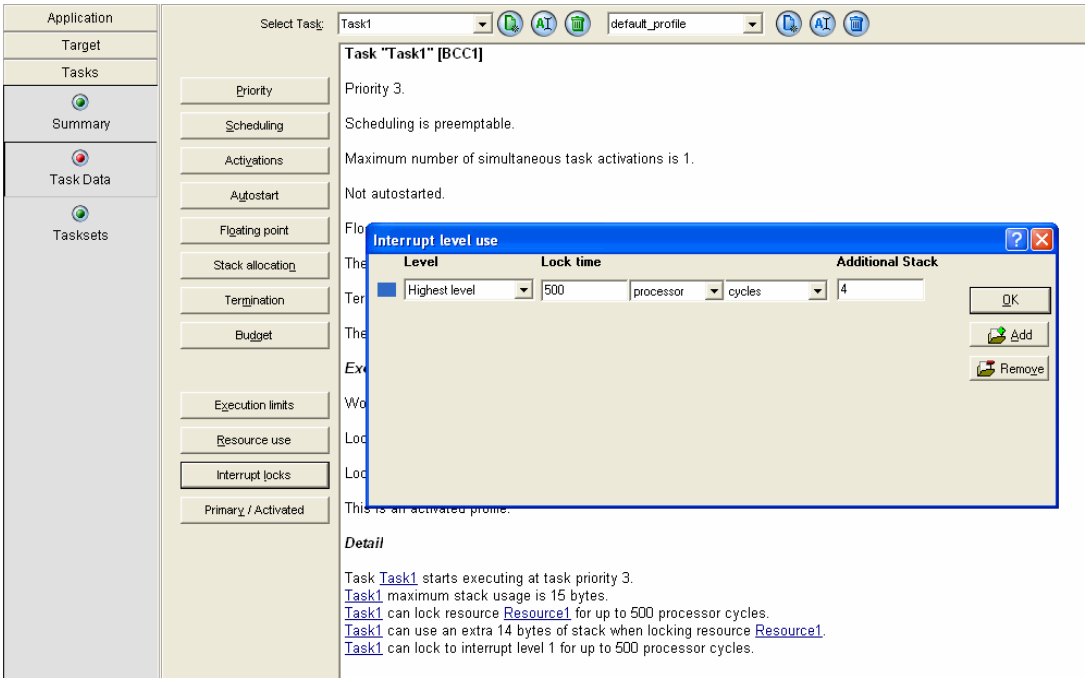


Figure 14:17 - Specifying Interrupt Lock Times

Code Example 14:1 shows that `Resource1` is held on two separate occasions, for 20 cycles and then 30 cycles. Only the longest time needs to be specified, but you can specify both if you want to.

```
TASK(Task1) {
  ...
  GetResource_Resource1();
  /* Held for 20 processor cycles. */
  ReleaseResource_Resource1();
  ...
  GetResource_Resource1();
  /* Held for 30 processor cycles. */
  ReleaseResource_Resource1();
  TerminateTask();
}
```

Code Example 14:1 - Occupying Resources

It is better if each separate period is specified, even though you only need to specify the longest single execution time. If you specify each period separately it will improve the clarity and will help with the maintenance of the configuration file. No single locking time can exceed the task's execution time.

You do not need to distinguish whether or not resource requests are nested. RTA-OSEK Planner takes account of this automatically during analysis.

For each resource or interrupt that is held, you can specify additional stack usage information. It is better if you enter the stack usage figures for each resource or interrupt. RTA-OSEK Planner cannot take into account any possible stack overlays if you only specify a single stack usage figure.

If RTA-OSEK Planner knows about stack overlays it can reduce stack usage when resources are held or interrupts are disabled.

Figure 14:18 shows the stack space required for `Task1` during its execution.

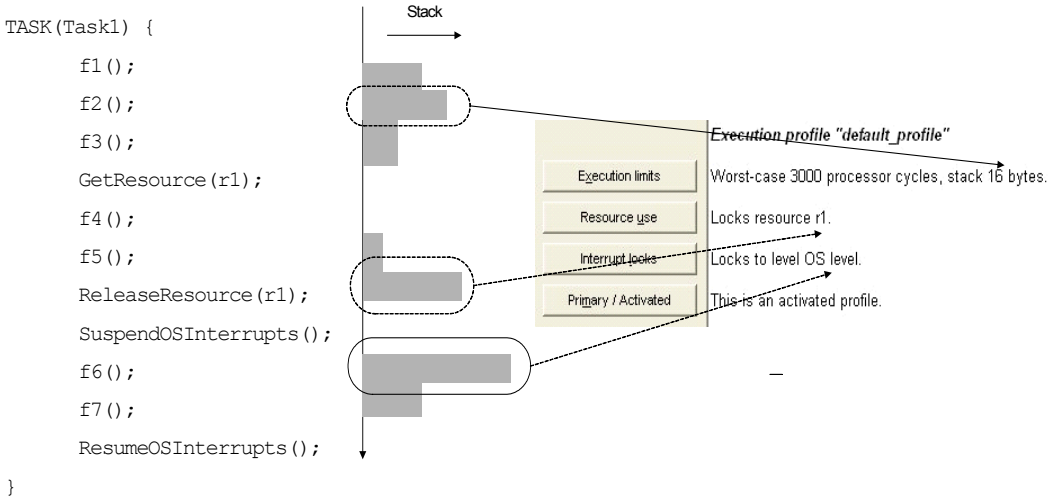


Figure 14:18 - Stack Space Required for Task1

To get a more accurate figure you will need to make a number of measurements. These measurements are required to determine the worst-case stack usage. Knowledge of the application functions will be needed to determine at which point in the task the most stack is consumed.

The measurements that you will need are:

- The worst-case stack usage from the first machine-code instruction in the task or ISR entry to any point, excluding places where code executes with resources held or where interrupts are disabled or suspended. This figure is the stack usage for the task or ISR.
- The worst-case stack usage from where a resource is held (for each held resource).
  This figure is the stack usage for the resource.
- The worst-case stack usage from where interrupts are suspended or disabled (for each interrupt level).
  This figure gives the stack use for each interrupt lock.

14.4 Target Specific Timing Information

The timing information that you have looked at, so far, has been for your application. To provide an accurate analysis, however, RTA-OSEK Planner needs to know information about the timing and operation of your target hardware.

- System timings.
  These are the execution times of aspects of RTA-OSEK Component, such as task entry and exit times.
- Interrupt recognition time.
  This is the maximum delay caused by the hardware before the first instruction of an interrupt can be acted upon.

- Arbitration ordering.
  This is the order in which interrupts of the same priority are processed.

Interrupt recognition time and arbitration ordering are target specific. The system timings depend on how your application makes use of certain target specific features.

Normally you will require some knowledge of how the application will be implemented on the target. This information is not always available during the early stages of design. When this happens, reasonable assumptions will have to be made and 'real' data will need to be substituted whenever it becomes available.

14.4.1 System Timings

In order to provide accurate timing analysis, RTA-OSEK must be told about how to account for operating system overheads. System timing data describes how many processor cycles particular operating systems take.

Because of the wide variety of possible target platforms and implementations, the best approach to measuring system timing information is to work in conjunction with ETAS' Engineering Services – consult us for details of how you can determine this information on your intended hardware platform.

Important: System timing information is specific to a particular hardware configuration. If you change your hardware or locate the application in a different memory area (by moving from on-chip to off-chip ROM, for instance) the system timings will need to be measured again. The values may also differ if you change the characteristics of an application by, for example, adding an extended task or an alarm.

System timing values must be generated to perform accurate analysis. If you cannot generate these values you will need to supply a set of plausible system timings. You could do this, for example to scope the timing behavior of a proposed system early in the development lifecycle. If you do not provide any set values, RTA-OSEK Planner will assume that they are zero.

14.4.2 Interrupt Recognition Time

The interrupt recognition represents the maximum time during which an interrupt will not be recognized by your target hardware. This is a single value and is entered in terms of CPU cycles.

Interrupt recognition time is usually at least equal to the execution time of the longest instruction (unless lengthy instructions can be interrupted part way through). Have a look at the *RTA-OSEK Binding Manual* and the manufacturers' data book for your target to find out how to obtain this information.

Figure 14:19 shows how the interrupt recognition time is specified.

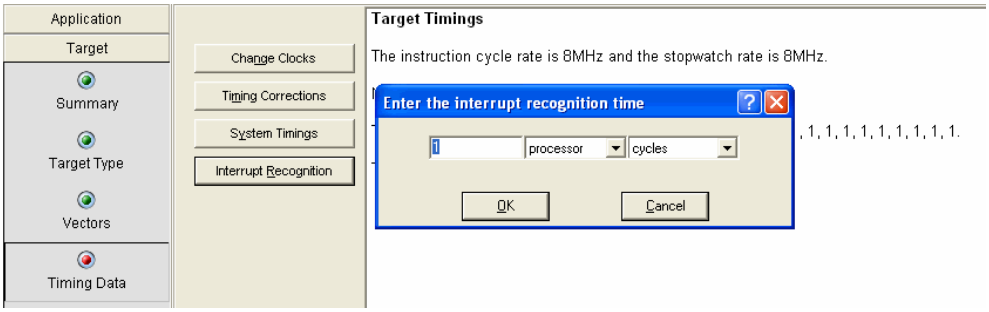


Figure 14:19 - Specifying the Interrupt Recognition Time

Interrupt recognition time is treated as blocking time by the analysis. This means that, for the entire duration of the interrupt recognition time, the processor will be executing instructions of a (soon to be interrupted) task, as if no interrupt had occurred. You must make sure that you do not classify interrupt handling overhead as interrupt recognition time.

14.4.3 Interrupt Arbitration

When ISRs share an interrupt priority level, you will have to enter an interrupt arbitration order. The arbitration order is the sequence in which interrupts of the same priority are serviced if several are pending at the same time. You can usually find this information in the data book for your target processor. The arbitration ordering allows RTA-OSEK Planner to determine interrupt blocking correctly for the specified interrupts. In Figure 14:20, `Bursting`, `Timer1` and `Timer2` share interrupt priority level 1. If all three interrupts are pending simultaneously, the RTA-OSEK Planner knows that `Bursting` will be processed first, followed by `Timer1` and finally `Timer2`.

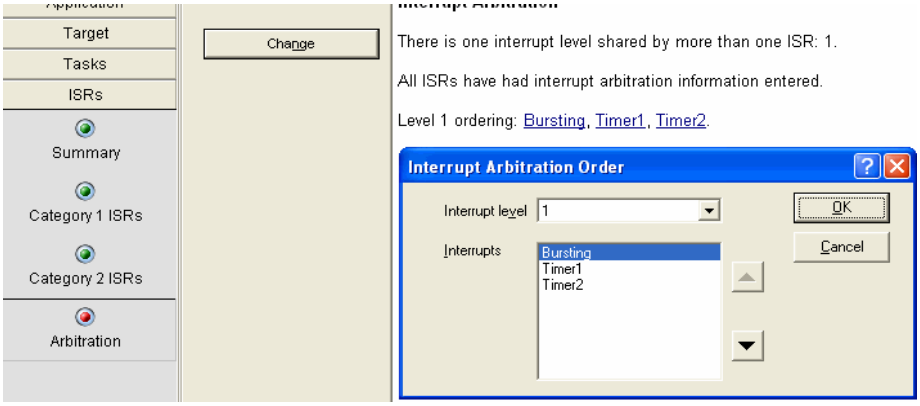


Figure 14:20 - Interrupt Arbitration Order

Advanced Modeling of the Application for RTA-OSEK Planner

14.5 Analyzing Alarms

When an application uses a counter and a series of alarms to implement a sequence of task activations, RTA-OSEK Planner assumes that each alarm can be stopped and restarted independently. This ensures that the worst-case timing behavior of the alarms on the counter is accounted for.

However, if the alarms are autostarted and they are not modified at run-time, this model is unnecessarily pessimistic. You can reduce this pessimism by specifying that the counter has synchronized alarms.

Figure 14:21 shows you how to select the alarm synchronization setting in the RTA-OSEK GUI. You must make sure that synchronization is maintained.

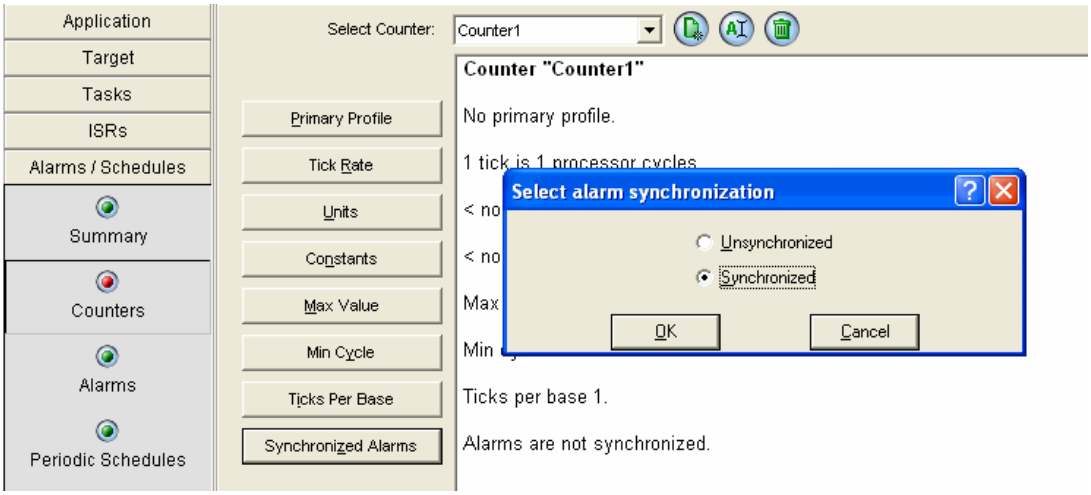


Figure 14:21 - Selecting Alarm Synchronization Settings

14.6 Specifying Multiple Execution Profiles

Tasks or ISRs can occur in several different execution contexts. If this happens, the pessimism in analysis can be reduced if multiple execution profiles are declared.

Multiple profiles are useful when tasks or ISRs have very short execution times when they are called in some contexts, but much longer execution times in others.

Code Example 14:2 shows how multiple execution profiles are specified.

```
if (Condition) {

  /* Short computation. */

} else {

  /* Long computation. */

}
```

Code Example 14:2 - Specifying Multiple Execution Profiles

Multiple execution profiles should be used where:

- An ISR services several different sources of interrupt and its execution behavior is different for each source.
- A task implements round-robin scheduling of its activities. For example, the first time it is activated, it performs A; the second time it performs B and so on.
- A task or ISR has different behavior depending on the current application mode.

When constructing multiple profiles for tasks or ISRs that can get resources or disable interrupts, you must consider whether or not each profile gets each specific resource.

In Code Example 14:3, `Task1` gets resource `Resource1` in one profile and disables interrupts in another profile.

```
TASK(Task1) {
  if (Condition) {
    ...
    GetResource_Resource1();
    ...
    ReleaseResource_Resource1();
    ...
  } else {
    ...
    DisableAllInterrupts();
    ...
    EnableAllInterrupts();
  }
  TerminateTask();
}
```

Code Example 14:3 - Using Multiple Profiles to Get Resources and Disable Interrupts

You only need to specify execution times and stack usage for the profiles where a resource is used or where an interrupt is disabled. You can enter zero execution times for the profiles that do not lock the resource or disable the interrupt.

If any information is missing you may receive inaccurate results from the analysis of your application.

Important: Make sure that where multiple profiles are defined, they are used in the model or are auto-activated (either directly or by an autostarted alarm). If a profile is omitted it will not be included in the analysis.

14.7 Looping and Retriggering Interrupt Behavior

In the systems that you have seen so far, the deadlines for the responses to stimuli are less than or equal to the arrival period of the stimuli. In these systems, each ISR must complete before it is next invoked.

Sometimes, though, the stimuli may arrive faster than they can be handled by the associated ISR (as the primary profile), but the response(s) can be generated after the next (or nth) arrival. You will see this if, for instance, you have to deal with the arrival of bursting messages over a network. In these cases the deadline for responding to the stimulus is longer than its period.

Usually, when this happens, you will need to provide some means of buffering the interrupts until they can be processed. This can be provided by some external interrupt control logic or, alternatively, your target microprocessor might support this. CAN controllers, for example, usually provide some hardware buffering for messages arriving over the network.

There are two ways that ISRs can deal with buffered interrupts:

- Looping.
  The outermost level of the ISR consists of a loop that checks whether unprocessed interrupts remain and, if so, repeats the processing.
- Retriggering.
  The final instruction(s) of the ISR checks whether unprocessed events remain and, if so, causes the interrupt to trigger the handler again.

Portability: The interrupt mechanism on your target platform affects the way that retriggering is achieved. Usually you must reassert the interrupt.

If you want RTA-OSEK Planner to take account of buffered behavior when analysis is performed, you must specify:

- That buffering is used.
- Whether the buffer is processed by retriggering the ISR or looping within it.
- The size of the buffer.

Figure 14:22 shows how the ISR buffering behavior is entered in the RTA-OSEK GUI.

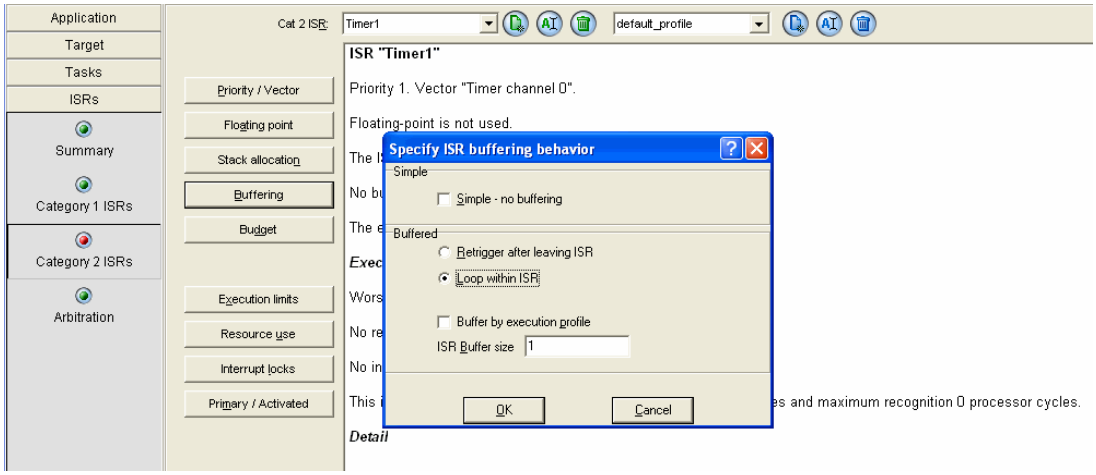


Figure 14:22 - Specifying Buffering Behavior

When buffered interrupts are handled by an ISR, you have seen that you can choose between retriggering and looping behavior. Normally retriggering behavior is recommended. There are some factors that will influence your choice of behavior.

Firstly, some hardware will not support retriggering behavior for interrupts. If this happens, a looping ISR must be used.

Secondly, a retriggering handler may be better if the interrupt that invokes the handler is at the same level as another interrupt in the system and if that other interrupt has a higher arbitration precedence. Higher arbitration precedence means that it will be handled first if both are pending. This may reduce the amount of blocking suffered by the other interrupt, which is important if your target only supports a single interrupt level.

Thirdly, a retriggering ISR could have a smaller execution time than a looping executable object when a single interrupt is processed. It doesn't matter that

a looping handler may be 'more efficient' when several events are handled in one invocation, because the analysis must assume worst-case behavior. This is where interrupts occur in a pattern that results in each one being handled by a separate invocation of the ISR.

Code Example 14:4 shows another example of multiple profiles. This ISR handles three interrupt sources detected by functions `Source1()`, `Source2()` and `Source3()`.

```
ISR(isr1) {

  if (Source1()) {

    /* Handle Source1. */

  } else if (Source2()) {

    /* Handle Source2. */

  } else if (Source3()) {

    /* Handle Source3. */

  }
}
```

Code Example 14:4 - Using Multiple Profiles

Three separate execution profiles are defined for the ISR in Code Example 14:4. They can be characterized by the results of the tests:

- `Source1()` returns true.
  The profile for this situation will include the worst-case execution time of the successful check of `Source1` handler code.
- `Source1()` returns `FALSE` and `Source2()` returns `TRUE`.
  The execution time for this profile will include the worst-case execution time required for the unsuccessful check of `Source1`, the successful check of `Source2` and the `Source2` handler code.
- `Source1()` and `Source2()` return `FALSE` and `Source3()` returns `TRUE`.
  The execution time of this profile is calculated in a similar way to the two profiles above.

Each of these profiles represents a complete path through the ISR (from the first instruction until the end of the final return instruction). Note that no profile exists for the case where all checks fail. This is because there is no way that the interrupt could be entered without one of the above conditions being true.

In some situations you will need a single handler that can handle multiple interrupt sources and each of these profiles must react to a different stimulus. In these cases, you should specify that the profiles are buffered by execution profile. This is shown in Figure 14:23.



Figure 14:23 - Buffering by Execution Profile

Code Example 14:4 should be modified to look like Code Example 14:5.

```
ISR(isr1) {
  do {

    if (Source1()) {

      /* Handle Source1. */

    } else if (Source2()) {

      /* Handle Source2. */

    } else if (Source3())     {

      /* Handle Source3. */
    }

  } while (interrupt_pending());
}
```

Code Example 14:5 - Buffering by Execution Profile

Tasks can also be buffered, but this is handled by allowing the task to have queued activation. In this case you don't need to provide any additional modeling information. RTA-OSEK Planner already knows the size of the buffer from configuration of RTA-OSEK Component. The task will be retriggered until the queue is empty.

14.8 Modeling Jitter

When an embedded real-time system is being analyzed it is important that timing figures relate to the stimuli and responses in the **embedding system**. RTA-OSEK Planner allows you to model the differences in time between the embedding system receiving stimuli and generating responses and the embedded system processing the stimuli and generating responses.

Sometimes there is a delay between the actual occurrence of a stimulus in the real-world and the notional release of the primary profile with the stimulus. There are many possible causes for this delay. It can be caused by, for example, slow hardware (such as some A/D converters) performing the detection.

The **recognition time** for stimuli is bounded by a minimum time representing the earliest release and maximum time representing the latest release. You can see an illustration of this in Figure 14:24.

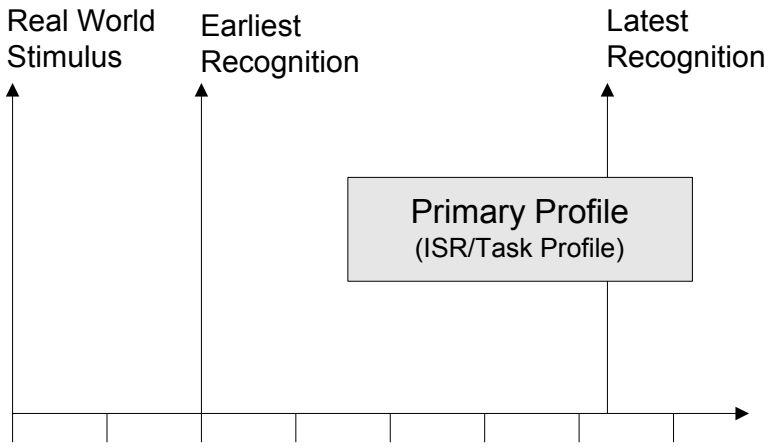


Figure 14:24 - Recognition Time

The difference between maximum and minimum recognition time is called the **input jitter**. The input jitter can be specified for each primary profile in the application.

For example, all real-world stimuli handled by a primary profiles may be subject to jitter of 50ns. This is the difference between the minimum response delay of 170ns and the maximum response delay of 220ns (220ns - 170ns = 50ns).

In a similar way, **output jitter** can be specified for responses. It can be used, for instance, to model situations that require an actuator to be driven when you must take account of hysteresis in the physical device.

Each response can be associated with a minimum and a maximum **response delay** that allows the latest time to be specified before a deadline that the response must be generated.

A response delay is illustrated in Figure 14:25.

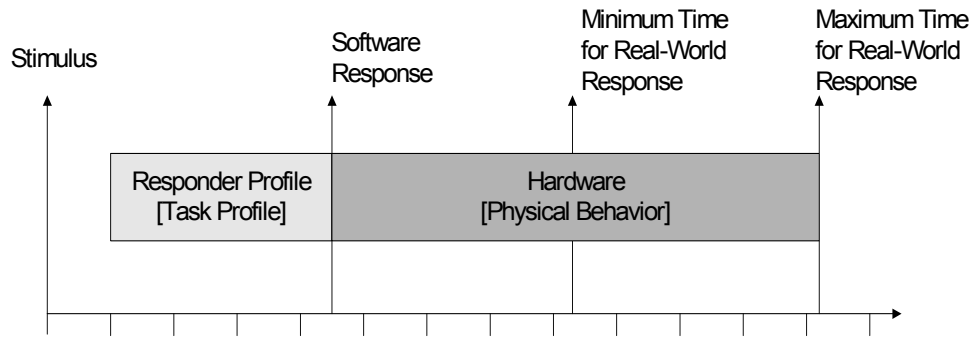


Figure 14:25 - Response Delay

In Figure 14:26, the response `EnergizeCoil` is specified with a deadline of 70ms, a minimum response delay of 20ms and a maximum response delay of 50ms.

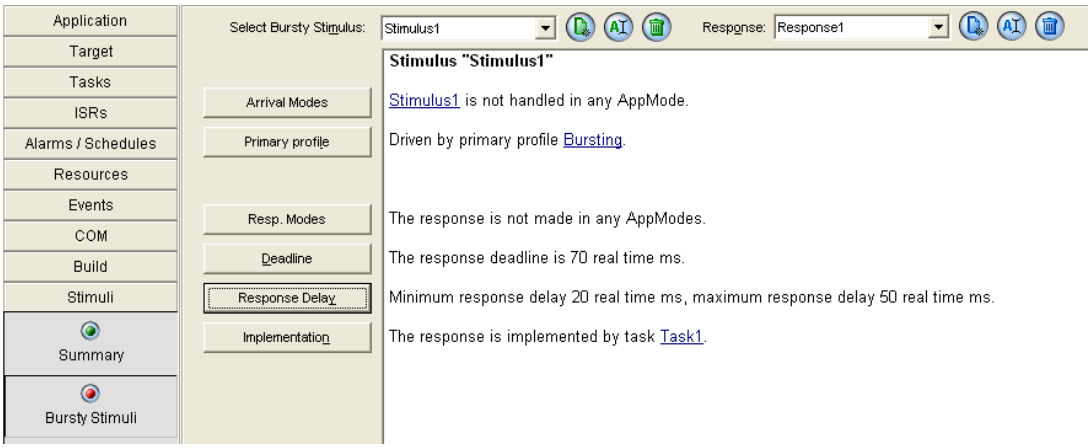


Figure 14:26 - Specifying a Deadline and a Response Delay

Important: A default value of 0 is assumed if response delays or minimum and maximum recognition times are not defined.

14.9 Modeling Planned Schedules

You saw earlier how planned schedules could be used to implement complex sequences of stimuli. If you are going to analyze your application for timing correctness, extra timing information must be supplied.

To change the behavior of an application, it is possible to modify planned schedules at run-time. You can modify delays, additional responses can be added to arrivalpoints and next clauses can be changed to switch specific responses in and out of the schedule.

To use this flexibility at run-time additional information about the worst-case behavior of the schedule must be provided. Worst-case behavior is represented by:

- The shortest delays between arrivalpoints.

- The maximum number of stimuli notionally triggered on each arrivalpoint.

Changes to the structure of a planned schedule are specified as **analysis overrides**. Additional information about the stimuli triggered from an arrivalpoint, including potential changes, is specified as indirectly activated stimuli.

Important: You will need to provide worst-case information about the schedule; otherwise RTA-OSEK Planner may indicate that the application is schedulable, even though modifications you make at run-time cause it to be unschedulable.

14.9.1 Specifying Analysis Overrides

You can use analysis overrides to tell RTA-OSEK Planner how the structure of the schedule may change at run-time. For timing analysis, when both analysis overrides and implementation details are present, the `delay` and `next` analysis attributes override the application attributes.

If the delay is changed at run-time, so that it is shorter than the implementation delay, then the shorter delay should be specified as an analysis override. Have a look at Figure 14:27.

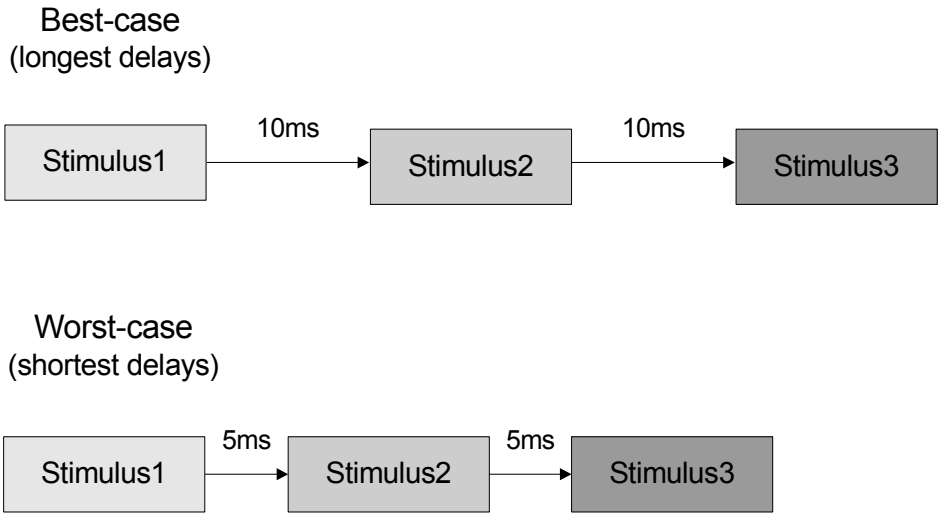


Figure 14:27 - Best Case and Worst Case

14.9.2 Indirectly Activated Stimuli

When you learnt about planned schedules you only specified the stimuli that were auto-activated on arrival. For analysis, RTA-OSEK Planner needs to know if any stimuli are indirectly activated.

An indirect activation is, for instance, when an auto-activated stimulus triggers a response that subsequently activates another task. Indirect activations need

to be specified when you want to model things like a task activating another task or additions to an arrivalpoint taskset at run-time.

Multiple indirectly activated stimuli can be specified for each arrivalpoint. The same stimulus can be both auto and indirectly activated. This, in fact, models the situation where a task chains itself.

For analysis there is no difference between a stimulus being directly activated and being indirectly activated. The observed behavior of the two situations is identical because there can be no upward activation in an analyzable system.

For timing analysis, RTA-OSEK Planner assumes that all auto and indirectly activated stimuli are triggered at once. For example, if an arrivalpoint auto-activates `Stimulus1` and indirectly activates `Stimulus2`, RTA-OSEK Planner assumes that the arrivalpoint releases the tasks generating the responses simultaneously. This represents the worst-case for the arrivalpoint.

14.10 Modeling Single-Shot Schedules

RTA-OSEK Planner assumes that a single-shot schedule only runs once in the entire run-time of the application.

A single-shot schedule may, however, be processed repeatedly by the system. If this happens, you will need to indicate that the implementation of the schedule is single-shot, but that it repeats for analysis purposes only.

A repeat for analysis purposes only is specified as an analysis override for the final arrivalpoint on the planned schedule. The next override must specify the first arrivalpoint on the schedule and the delay to next must specify the minimum time between successive activations of the schedule.

Have a look at Figure 14:28 to see how this is specified in the RTA-OSEK GUI.

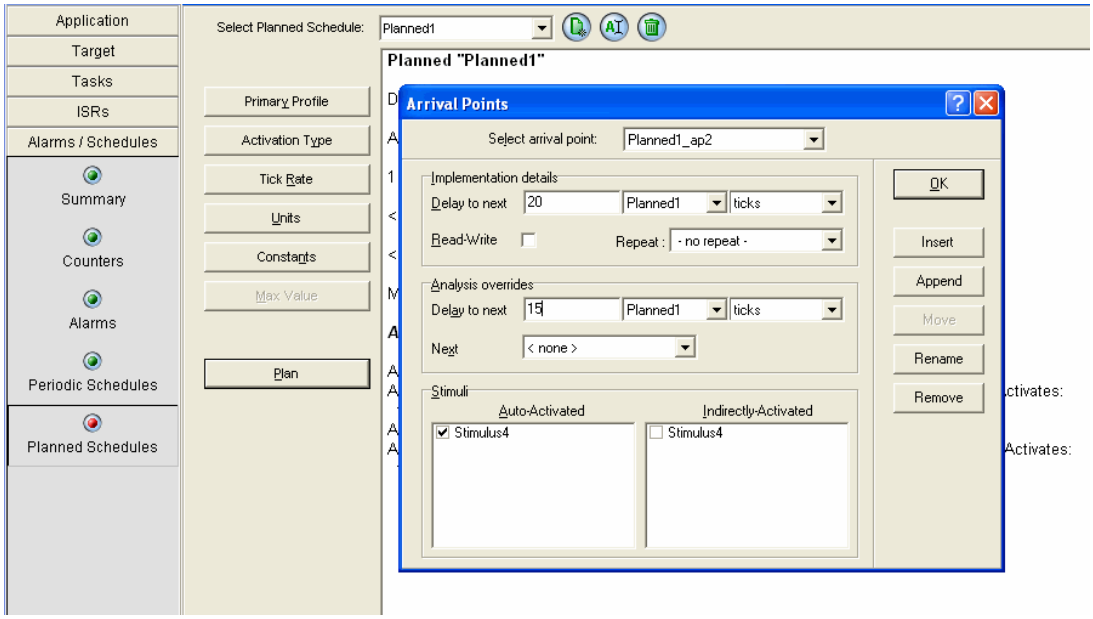


Figure 14:28 - Modeling a Single-Shot Schedule

Important: The value selected for the delay between subsequent repetitions of a single-shot schedule needs to be based upon knowledge of the application. Selecting a delay that is larger than the minimum delay may result in optimistic analysis. It could falsely indicate that a system is schedulable. If the value is too small, however, it may result in unnecessary pessimism.

14.11 Modeling with Extended Tasks

When you use extended tasks in your application you can only analyze the set of basic tasks that are of higher priority than all extended tasks. This is because extended tasks wait on events and RTA-OSEK Planner cannot analyze this.

You must still specify resource and interrupt locking times because this will affect the amount of blocking suffered by all higher priority basic tasks.

Important: If you want to use extended task behavior, but want to analyze your entire application, you should consider simulating extended tasks using the scheme that is outlined in the chapter on Tasks.

Modeling the Application for RTA-OSEK Planner Summary

- If you need to do analysis of your application then you must specify execution performance constraints for your stimulus/response model, worst-case execution times for each task or ISR and target timings.
- Performance constraints are specified as part of your stimulus/response model.

- Bursty stimuli are used to model simple cases where a primary profile captures a stimulus directly.
- Planned and periodic stimuli are used to model more complex cases where a primary profile drives a counter or schedule to generate stimuli.
- Each task and ISR in your application must have at least one profile that specifies execution information and whether the profile can be used in the capture or generation of a stimulus.
- You can reduce pessimism in the system by specifying, where possible, that alarms are synchronized.
- You can also reduce pessimism using multiple profiles for each task and ISR.
- Interrupt buffering can be modeled.
- If your timing needs to be correct with respect to your embedding system, you need to specify input and output jitter for each primary profile and each response respectively.
- Planned schedules must include analysis information to capture stimuli that trigger other stimuli and to capture changes to schedule behavior at run-time.

15 Performing Analysis

The RTA-OSEK GUI provides access to RTA-OSEK Planner for analyzing your application. There are 5 analysis options:

- Stack Depth analysis.
- Schedulability analysis.
- Sensitivity analysis.
- Best Task Priorities analysis.
- CPU Clock Rate analysis.

Stack depth, schedulability and sensitivity analysis are used to tell you about the memory usage and timing behavior of your application. Best task priorities and clock rate analysis suggest ways that your application can be optimized for either space or time.

To make use of analysis, you must specify execution time and stack space information. This is provided in an execution profile for each task and ISR in your application. If any of this information is not present, the analysis summary will tell you which parts are missing.

It is important to make sure that your RTA-OSEK Planner model accurately reflects your application. If you create tasks and ISRs that are not attached to a stimulus/response model, they will not be included in the analysis.

15.1 Stack Depth Analysis

If you have specified worst-case stack usage figures for each task and ISR in your application, then RTA-OSEK Planner can determine the worst-case stack usage for your application. Figure 15:1 shows how these values are gathered.

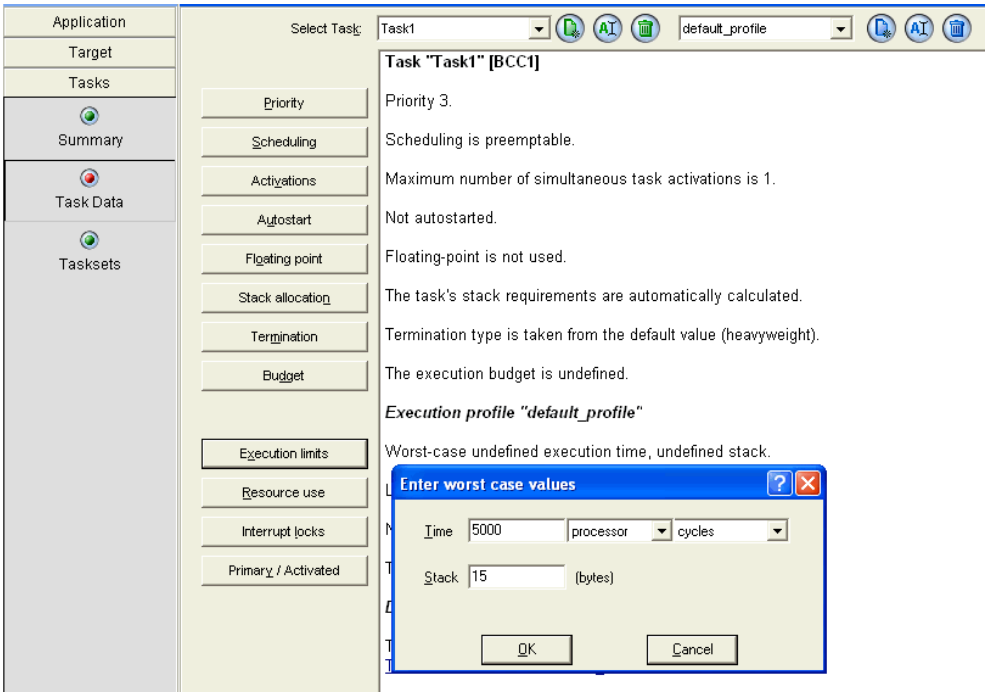


Figure 15:1 - Worst-Case Stack Usage

When you run the stack depth analysis, the stack analysis report appears on the workspace. There are two tabs that can be used to view the results of the analysis. The results are available in text format, as shown in Figure 15:2.

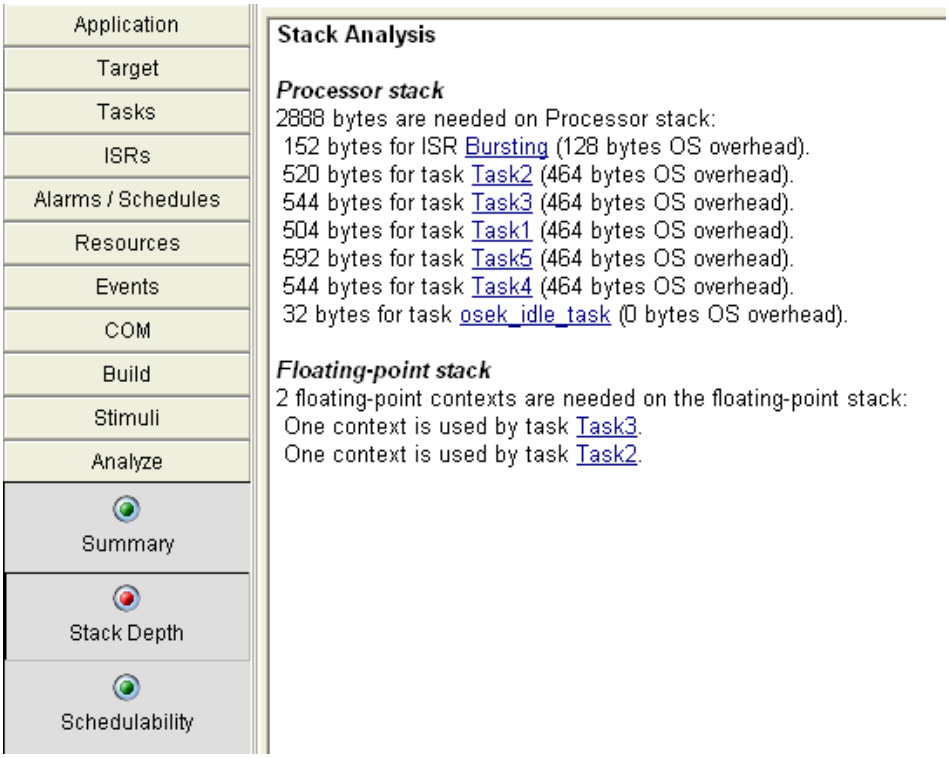

Application
Target
Tasks
ISRs
Alarms / Schedules
Resources
Events
COM
Build
Stimuli
Analyze
Summary
Stack Depth
Schedulability
Stack Analysis
Processor stack
2888 bytes are needed on Processor stack:
152 bytes for ISR Bursting (128 bytes OS overhead).
520 bytes for task Task2 (464 bytes OS overhead).
544 bytes for task Task3 (464 bytes OS overhead).
504 bytes for task Task1 (464 bytes OS overhead).
592 bytes for task Task5 (464 bytes OS overhead).
544 bytes for task Task4 (464 bytes OS overhead).
32 bytes for task osek_idle_task (0 bytes OS overhead).
Floating-point stack
2 floating-point contexts are needed on the floating-point stack:
One context is used by task Task3.
One context is used by task Task2.

Figure 15:2 - Stack Analysis Results in Text Format

You can also view the results on the Graphic tab. The analysis shows the maximum size of the stack in the workspace. An example is shown in Figure 15:3.

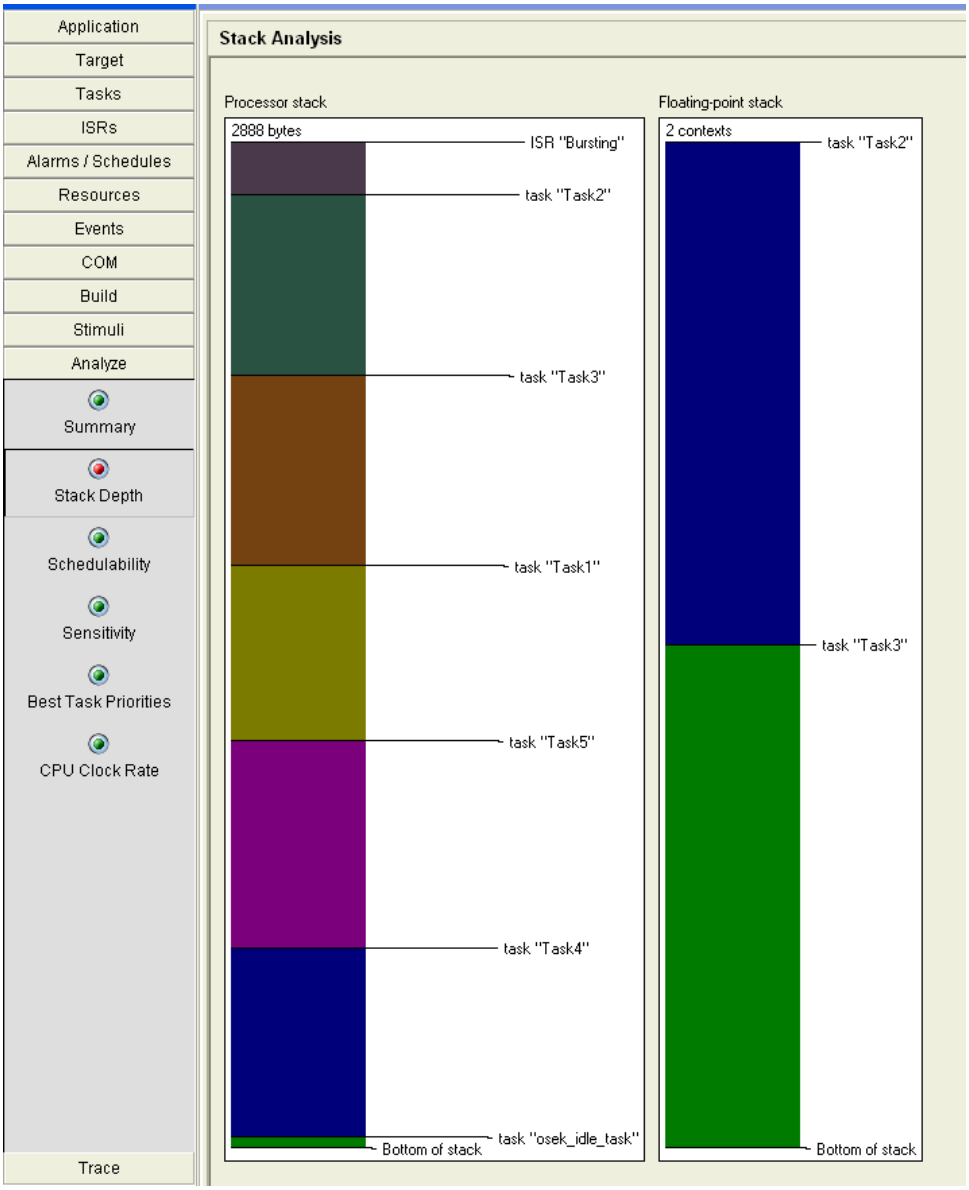


Figure 15:3 - Stack Analysis Results in Graphical Format

The analysis uses the following information when calculating the worst-case stack usage:

- Worst-case stack usage for each task and ISR profile.
- RTA-OSEK Component overheads for each task and ISR.
- Non-preemption information based on internal resources.
- Non-preemption information based on resources being held.
- Non-preemption information based on when interrupts are disabled.

The stack used in hook functions, callbacks or `GetStopwatch()` is not included in the calculation of the stack requirement for the application.

Normally, the application's total stack requirement is not changed by use of these functions. However, if you need to calculate the exact worst-case stack

usage of an application that uses hooks or callbacks and stack in `GetStopwatch()`, you may need to contact LiveDevices.

You will see that `GetStopwatch()` normally returns just the contents of a timer register. It does not place anything on the stack.

15.1.1 Floating-Point Context Saving

When tasks or ISRs use floating-point, RTA-OSEK Component saves a floating-point context whenever necessary. RTA-OSEK uses the architecture of your application to work out the maximum number of floating-point contexts that must be saved at run-time.

If, for example, two tasks use floating-point and share an internal resource they will never preempt each other. This means that they will never need to save any floating-point context. The RTA-OSEK GUI shows the maximum number of floating-point contexts required in your application.

15.1.2 Minimizing Stack Usage

You might find that the stack space required by your application is greater than the space available on your target hardware. If this happens, there are a number of things you can do to minimize application stack space.

- Share an internal resource between tasks. This means that the tasks will never preempt each other, so they will never require space on the stack at the same time. This effectively overlays the stack usage of all the tasks that share the internal resource. This can be done automatically using best task priorities analysis (see Section 15.4).
- If a task calls a function that uses a lot of stack space, you can get a resource around the function call. You can then share that resource with higher priority tasks (the higher priority tasks do not need to use the resource). This prevents the higher priority tasks from preempting the task calling the function while it uses a lot of stack space. This reduces overall usage.
- Interrupts can be disabled or combined resources can be used in the previous method. This allows you to prevent interrupts occurring while the function is being called. In this case, however, you will pay a penalty in interrupt latency.

15.2 Schedulability Analysis

Schedulability analysis is used to work out whether each response can be generated before its deadline. The deadline can be either explicit or implicit.

An explicit deadline is when, for example, you specify that a task must generate a response no more than 5ms after it is released. An example of an implicit deadline is when a task must finish before it is next made *ready* to run. For example, when a basic unqueued task is activated every 20ms, it must complete before it is next released. This means that it has an implicit deadline of 20ms.

Schedulability analysis calculates the worst-case response time for each execution profile in your application and then determines if these response times are less than the associated deadlines. The RTA-OSEK GUI shows the results of schedulability analysis. You can see an example in Figure 15:4.

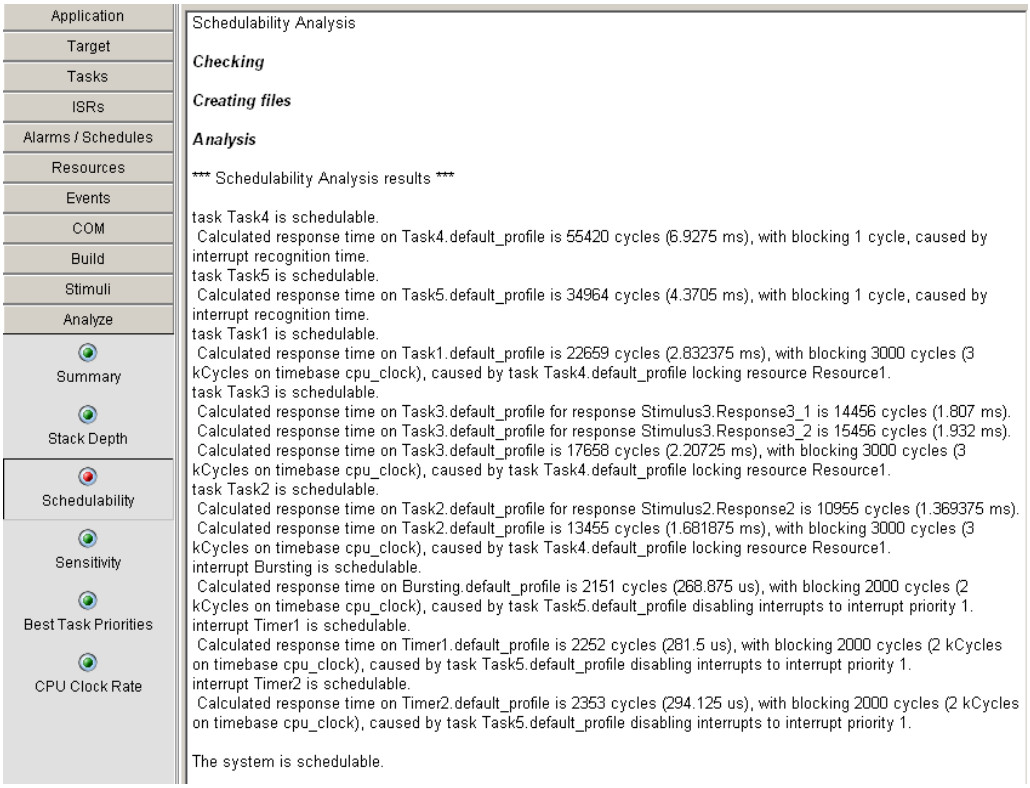

Application
Target
Tasks
ISRs
Alarms / Schedules
Resources
Events
COM
Build
Stimuli
Analyze
Summary
Stack Depth
Schedulability
Sensitivity
Best Task Priorities
CPU Clock Rate
Schedulability Analysis
Checking
Creating files
Analysis
*** Schedulability Analysis results ***
task Task4 is schedulable.
Calculated response time on Task4.default_profile is 55420 cycles (6.9275 ms), with blocking 1 cycle, caused by interrupt recognition time.
task Task5 is schedulable.
Calculated response time on Task5.default_profile is 34964 cycles (4.3705 ms), with blocking 1 cycle, caused by interrupt recognition time.
task Task1 is schedulable.
Calculated response time on Task1.default_profile is 22659 cycles (2.832375 ms), with blocking 3000 cycles (3 kCycles on timebase cpu_clock), caused by task Task4.default_profile locking resource Resource1.
task Task3 is schedulable.
Calculated response time on Task3.default_profile for response Stimulus3.Response3_1 is 14456 cycles (1.807 ms).
Calculated response time on Task3.default_profile for response Stimulus3.Response3_2 is 15456 cycles (1.932 ms).
Calculated response time on Task3.default_profile is 17658 cycles (2.20725 ms), with blocking 3000 cycles (3 kCycles on timebase cpu_clock), caused by task Task4.default_profile locking resource Resource1.
task Task2 is schedulable.
Calculated response time on Task2.default_profile for response Stimulus2.Response2 is 10955 cycles (1.369375 ms).
Calculated response time on Task2.default_profile is 13455 cycles (1.681875 ms), with blocking 3000 cycles (3 kCycles on timebase cpu_clock), caused by task Task4.default_profile locking resource Resource1.
interrupt Bursting is schedulable.
Calculated response time on Bursting.default_profile is 2151 cycles (268.875 us), with blocking 2000 cycles (2 kCycles on timebase cpu_clock), caused by task Task5.default_profile disabling interrupts to interrupt priority 1.
interrupt Timer1 is schedulable.
Calculated response time on Timer1.default_profile is 2252 cycles (281.5 us), with blocking 2000 cycles (2 kCycles on timebase cpu_clock), caused by task Task5.default_profile disabling interrupts to interrupt priority 1.
interrupt Timer2 is schedulable.
Calculated response time on Timer2.default_profile is 2353 cycles (294.125 us), with blocking 2000 cycles (2 kCycles on timebase cpu_clock), caused by task Task5.default_profile disabling interrupts to interrupt priority 1.
The system is schedulable.

Figure 15:4 - Results of Schedulability Analysis

The results of analysis can also be viewed graphically. An example is shown in Figure 15:5.

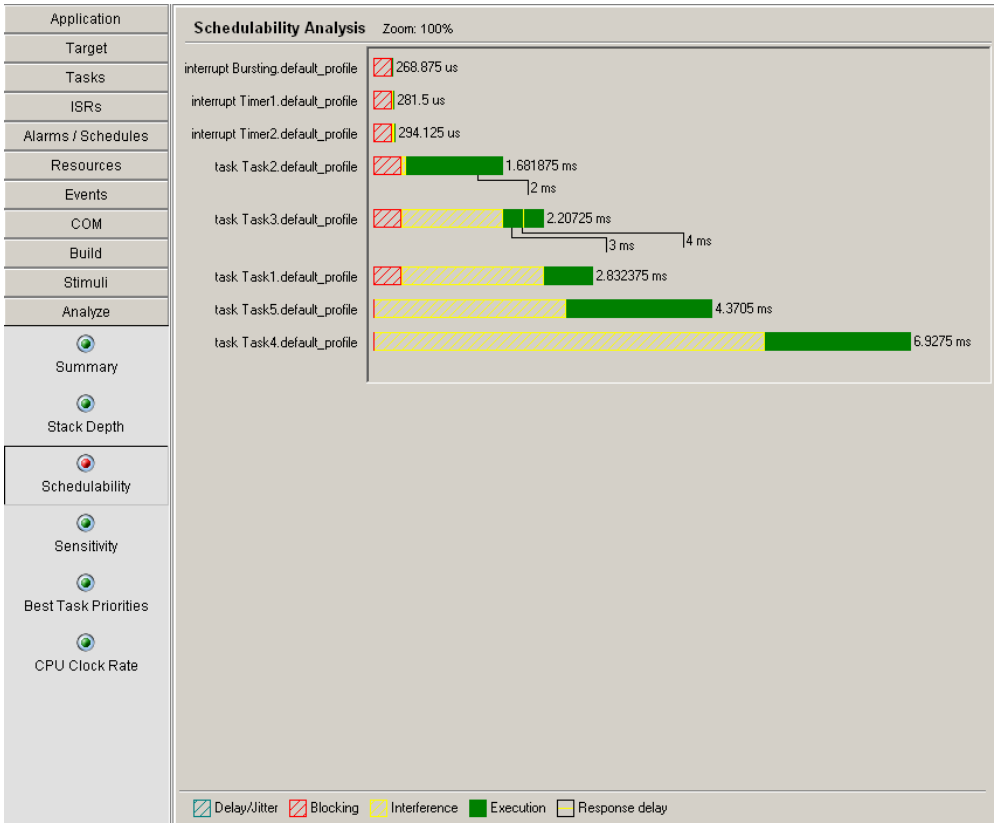


Figure 15:5 - Graphical Results of Schedulability Analysis

Each bar in the graphical analysis report shows the response time for the execution profile. Each bar has up to 5 sections:

- Delay/Jitter.
  This is the maximum amount of time that the primary profile, which responds to a stimulus, takes to recognize the stimulus. This is specified in the **Primary or Activated Profile** dialog for the appropriate primary profile.
- Blocking time.
  This is the amount of time that the execution profile is prevented from executing by a *lower priority* profile that holds a shared resource or has disabled interrupts.
- Interference.
  This is the amount of time that the execution profile is prevented from running by *higher priority* tasks or ISRs. This is the total amount of time that the profile is preempted during execution.
- Execution time.
  This is the worst-case execution time that you specified for the execution profile.
- Response delay.
  This is the time from the response being generated by the software to it being observable in the external environment. This is usually only specified when the response drives some external hardware.

In the graphical display, a 'tool-tip' will appear (shown in Figure 15:6) when you hover the mouse pointer over the execution time component for each task or ISR. This tells you the actual times for each of these components.



Figure 15:6 - Tool-tip Showing the Actual Times

If you have specified explicit deadlines for responses, these are represented on the bar as small tags with the deadline specified. Implicit deadlines are not shown.

In addition to this information, the textual output will tell you about queued activation counts, buffered interrupts and the parts of the system that contribute to the blocking time.

For any analyzable system, schedulability analysis reports that a system is either **schedulable** or **not schedulable**. If a system is schedulable, this means that all tasks or ISRs in the system will always meet all of their deadlines.

If the system is reported to be not schedulable, this is because either:

- Some responses in the system cannot be generated before their deadlines. The system is **unschedulable**.
- It is not possible to determine whether or not the system is schedulable. The system has **indeterminate schedulability**.

You'll see later what you should do if your application has indeterminate schedulability or is unschedulable.

You can also use sensitivity analysis to direct you to the parts of the system that would benefit from the most attention. You'll learn about this in Section 15.3.

Important: You should never change the configuration of your application without first validating those changes against detailed system analysis. RTA-OSEK Planner can only assess the timing correctness using the information that you provide.

15.2.1 Unschedulable Systems

Systems can be unschedulable for a variety of reasons:

- A task or ISR cannot complete before its next release.
- A task or ISR does not generate a response before a specified deadline.
- An ISR with looping or retriggering behavior exceeds the buffer limit or a basic task with queued activation exceeds the queue limit.
- The system exceeds 100% CPU utilization.

You'll now see each of these situations in more detail. You will also find out how you can modify your system so that it can be scheduled.

15.2.1.1 Task or ISRs Cannot Complete Before Next Release

If a task or ISR cannot complete before its next release, RTA-OSEK Planner will generate the message in Figure 15:7. In this example, `Task2` is not schedulable.

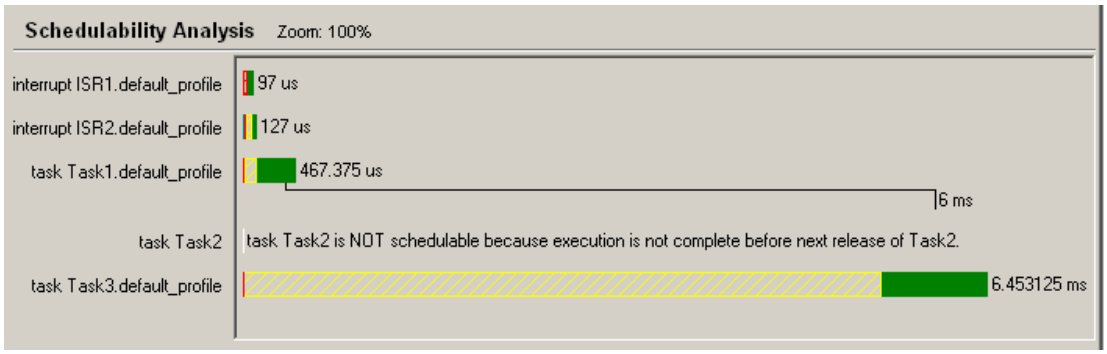


Figure 15:7 - Task Execution not Complete before the Next Release of the Task

There are a number of approaches you can use to make this type of system schedulable:

- Reduce the execution time of the task or any other higher priority tasks or interrupts. By reducing execution times you can reduce the amount of interference suffered by lower priority tasks and interrupts.
- If the task or any higher priority tasks or interrupts are periodic, their periods can be increased. If the task being adjusted has multiple offsets, these can be altered.
- Introduce queued activation for tasks or buffer interrupts to ensure that activations made whilst the tasks or ISRs are executing are not lost.
- Other tasks within the system may be making a specific task unschedulable. You could use best task priorities analysis (which you'll find out more about in Section 15.4) to see if a different priority ordering will make the system schedulable.
- If the unschedulable task or ISR shares a resource with lower priority task or ISR then you could try reducing the amount of time for which the resource is held by these tasks and ISRs. This reduces blocking times and may make the task schedulable.

These measures can also be used where systems are found not to be schedulable for other reasons.

15.2.1.2 A Task or ISR Cannot Meet its Deadline

If a task or ISR cannot meet a deadline, this results in an unschedulable system. There are two scenarios for detection:

- **Non-queued task activation and non-buffered ISRs**. A specific profile is found to be not schedulable because the deadline has been exceeded. In this case, other profiles of the same task or ISR might also be found to be schedulable (or unschedulable).

- **Queued task activations (BCC2 tasks) and buffered ISRs**. The task or ISR is not schedulable because the deadline of one of its profiles has been exceeded. In this case, none of the other profiles of the task or ISR will be found to be schedulable.

When a task or ISR is not schedulable because its deadline cannot be met, you can try to:

- Increase the deadline.
- Move the response generation code earlier in the program. This shortens the amount of time that the task or ISR must execute to generate the response.
- Use the suggestions for unschedulable systems that are mentioned above.

15.2.1.3 Queuing Task Activations and Buffered Interrupts Exceeded

Sometimes a system will not be schedulable because the queue for queued task activations is not long enough to hold the maximum number of activations that can occur whilst the task is *running*. Similarly, for interrupts that are buffered, the number of interrupts that need to be buffered may exceed the buffer size.

Figure 15:8 shows RTA-OSEK Planner output where the number of buffered activations for an interrupt is too small.

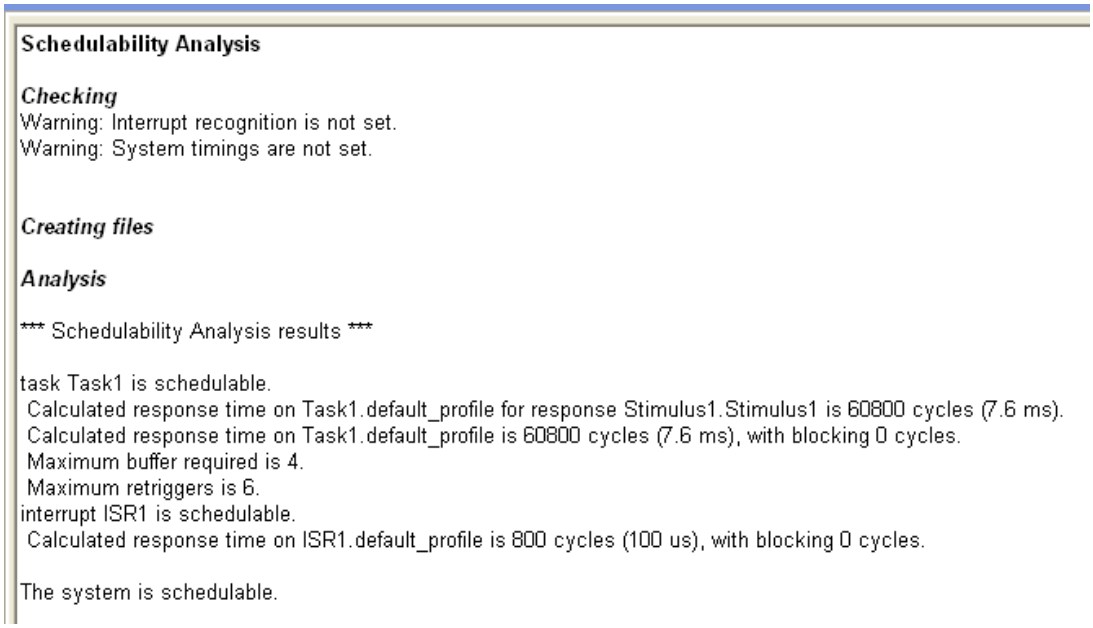

Schedulability Analysis
Checking
Warning: Interrupt recognition is not set.
Warning: System timings are not set.
Creating files
Analysis
*** Schedulability Analysis results ***
task Task1 is schedulable.
Calculated response time on Task1.default_profile for response Stimulus1.Stimulus1 is 60800 cycles (7.6 ms).
Calculated response time on Task1.default_profile is 60800 cycles (7.6 ms), with blocking 0 cycles.
Maximum buffer required is 4.
Maximum retriggers is 6.
interrupt ISR1 is schedulable.
Calculated response time on ISR1.default_profile is 800 cycles (100 us), with blocking 0 cycles.
The system is schedulable.

Figure 15:8 - Results of Schedulability Analysis Showing Buffered Activation

There are two things that may be causing this problem:

- The tasks or interrupts are being activated more frequently than they can be handled.
- The buffer sizes are too small.

Systems, which are unschedulable for these reasons, can be made schedulable. You can try to:

- Change the priorities to ensure that the task can handle the inputs at a required rate. If you do this, try using best task priorities analysis (see Section 15.4).
- Decrease the period of the task or ISR.
- Increase the buffer size.
- Decrease the execution time of the task.

If the analysis is repeated with a queue size or buffer size larger than needed, RTA-OSEK Planner reports the buffer size required to make the system schedulable.

15.2.1.4 Utilization Greater Than 100%

RTA-OSEK Planner may report that utilization is greater than 100%. This means that your application requires more time to execute than the time available on your target hardware. Figure 15:9 shows RTA-OSEK Planner report for a system with greater than 100% utilization.

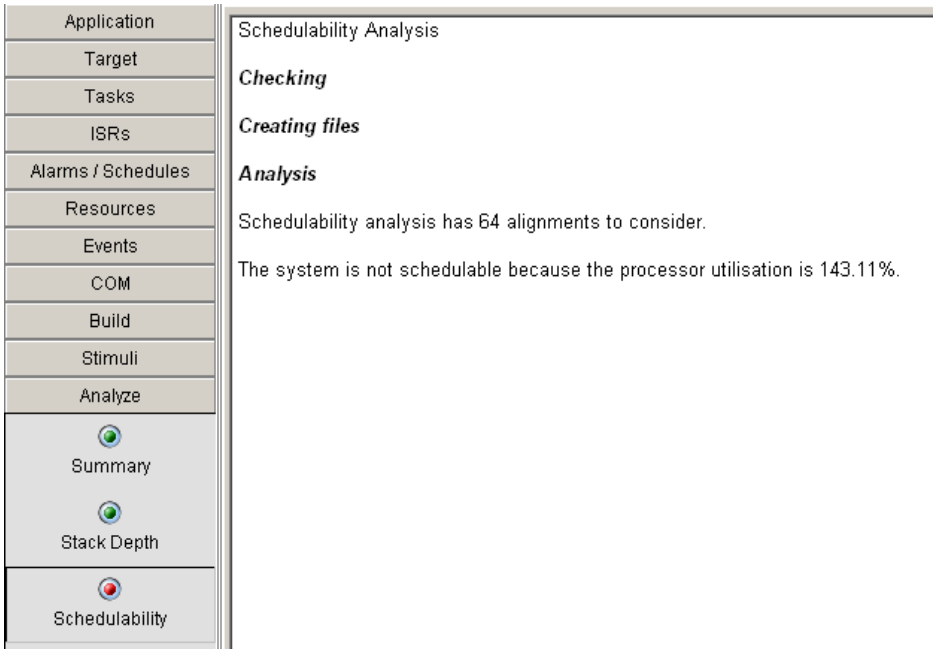


Figure 15:9 - Utilization of the Process is Greater than 100%

There are a number of strategies you can use to rectify this problem:

- Increase the CPU speed.
  This may be possible by specifying a faster part in your hardware design.
- Reduce the execution times for tasks and ISRs.
- Increase the periods for system stimuli.

15.2.2 Indeterminate Schedulability

Indeterminate schedulability occurs when:

- The busy period is too long to analyze.
- There is indeterminate blocking from lower priority tasks.
- Tractable analysis cannot determine schedulability.

You can use the methods described in Section 15.2.1 to try to fix indeterminate schedulability.

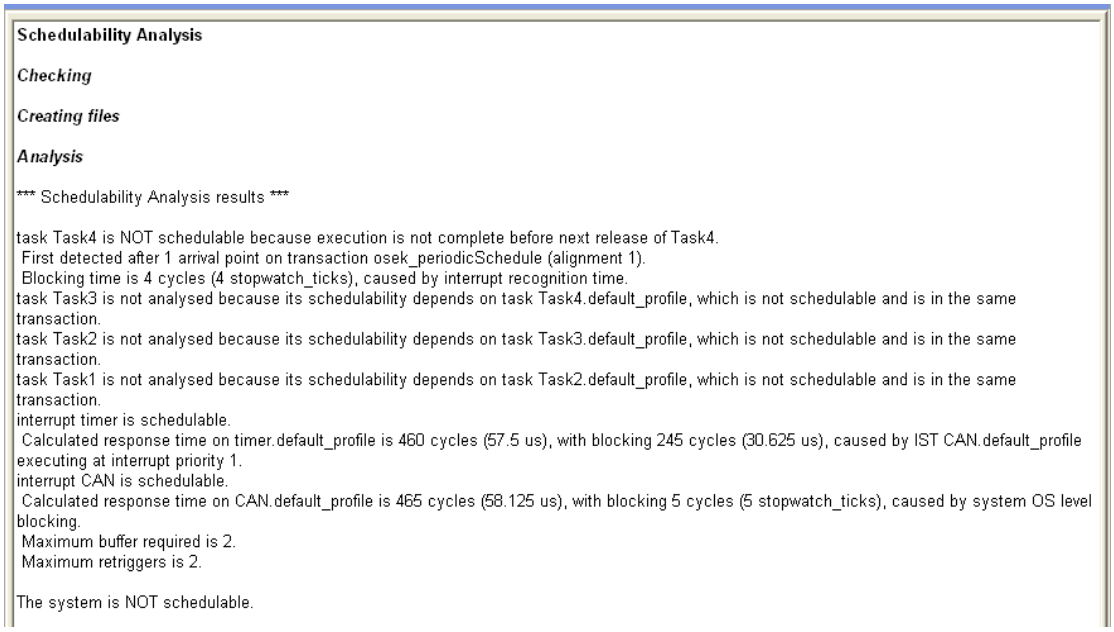
15.2.2.1 Busy Period too Long to Analyze

The busy period is the sum of the time that a task is in the *ready* or *running* state and the maximum recognition time.

RTA-OSEK Planner considers that the busy period is too long to analyze when it exceeds 2^{32} instruction cycles. In a valid real-world system, it is very unlikely that you will reach this value.

15.2.2.2 Indeterminate Blocking from Lower Priority Tasks or ISRs

If there is indeterminate blocking from lower priority tasks then RTA-OSEK Planner cannot calculate the response times of any tasks that share resources with the lower priority task. Sample output from RTA-OSEK Planner is shown in Figure 15:10.



Schedulability Analysis

Checking

Creating files

Analysis

*** Schedulability Analysis results ***

task Task4 is NOT schedulable because execution is not complete before next release of Task4.
First detected after 1 arrival point on transaction osek_periodicSchedule (alignment 1).
Blocking time is 4 cycles (4 stopwatch_ticks), caused by interrupt recognition time.
task Task3 is not analysed because its schedulability depends on task Task4.default_profile, which is not schedulable and is in the same transaction.
task Task2 is not analysed because its schedulability depends on task Task3.default_profile, which is not schedulable and is in the same transaction.
task Task1 is not analysed because its schedulability depends on task Task2.default_profile, which is not schedulable and is in the same transaction.
interrupt timer is schedulable.
Calculated response time on timer.default_profile is 460 cycles (57.5 us), with blocking 245 cycles (30.625 us), caused by IST CAN.default_profile executing at interrupt priority 1.
interrupt CAN is schedulable.
Calculated response time on CAN.default_profile is 465 cycles (58.125 us), with blocking 5 cycles (5 stopwatch_ticks), caused by system OS level blocking.
Maximum buffer required is 2.
Maximum retriggers is 2.

The system is NOT schedulable.

Figure 15:10 - Results of Schedulability Analysis Showing Indeterminate Blocking

Fixing this situation requires an incremental approach. Make the lower priority task schedulable first then iteratively apply schedulability analysis until your system is schedulable.

15.2.2.3 Tractable Analysis cannot Determine Schedulability

When driven by the RTA-OSEK GUI, RTA-OSEK Planner is set to use exact analysis by default (analysis depth 9). If you have configured the analysis depth for tractable analysis (analysis depth 1) then tractable analysis may not be able to determine schedulability.

Tractable analysis uses two approximations, a schedulability test and an unschedulability test. These approximations divide systems into categories.

Those that are:

- Definitely schedulable.
- Definitely unschedulable.
- Indeterminately schedulable.
  In this case the analysis depth should be set to 9. The schedulability analysis should then be re-run to find out if an indeterminately schedulable system is schedulable or not.

15.3 Sensitivity Analysis

Sensitivity analysis is used to explore the boundaries for your system. It allows you to answer questions like:

- What variation of clock speed is feasible?
- What is the maximum execution time allowed for a task or ISR?
- How long can I get a particular resource for?
- How long can I disable interrupts for?
- Can I vary the execution time for a response and still meet my deadline?

Sensitivity analysis allows you to determine what changes may make an unschedulable system schedulable. You might be able to optimize task and ISR execution times and a small reduction may be enough to make the system schedulable.

Alternatively, if you want to add additional functionality to an existing application then you can use sensitivity analysis to investigate how much headroom is available on which tasks or ISRs.

The sensitivity of the tasks and ISRs is considered against the following factors:

- Sensitivity to processor clock speed.
- Sensitivity to execution times.
- Sensitivity to resource and interrupt holding times.
- Sensitivity to response deadlines.

Figure 15:11 shows sensitivity analysis performed on a sample system.

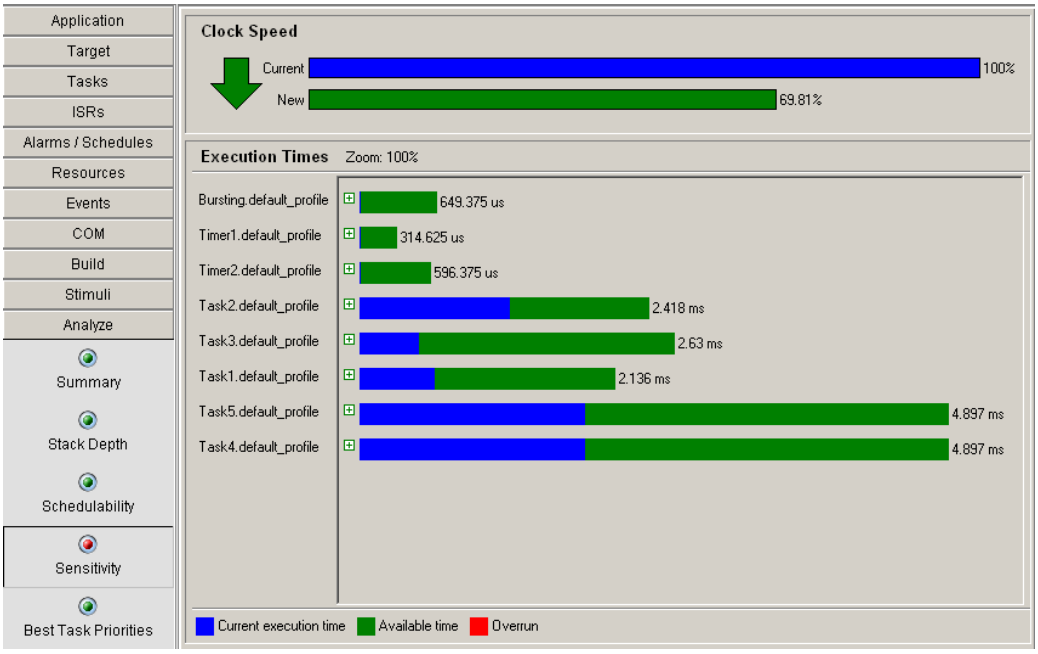


Figure 15:11 - Viewing Sensitivity Analysis Results in Graphical Format

The upper part of the sensitivity workspace shows the minimum clock speed required for the system to be schedulable. The sensitivity for execution times, lock times and response generation times is shown in the lower part of the workspace.

The results are displayed in color. Blue indicates current times, green indicates maximum available time and red indicates overrun.

The figures at the end of each bar give the maximum execution time, resource holding time or interrupt disabling time. You can use these values and still have a system that is schedulable.

The figures that are displayed are generally mutually exclusive, which means that you can implement any one of them and the system will become 'just' schedulable.

The sensitivity analysis report is also available in text format. You can see an example of this in Figure 15:12.

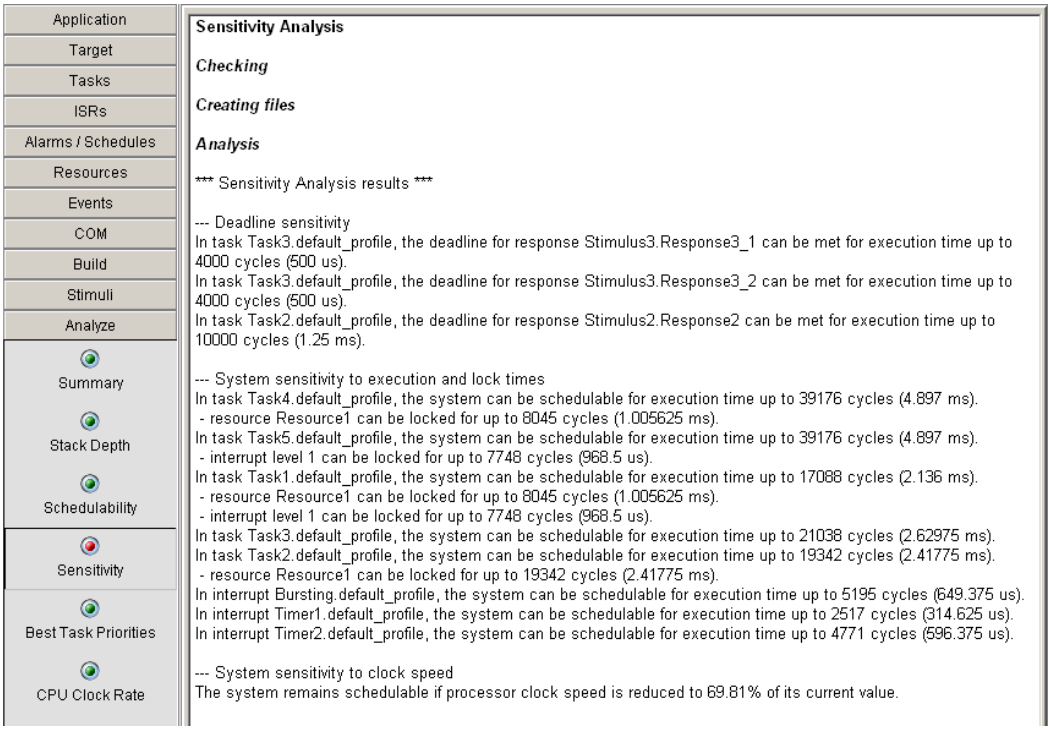

Application
Target
Tasks
ISRs
Alarms / Schedules
Resources
Events
COM
Build
Stimuli
Analyze
Summary
Stack Depth
Schedulability
Sensitivity
Best Task Priorities
CPU Clock Rate
Sensitivity Analysis
Checking
Creating files
Analysis
*** Sensitivity Analysis results ***
--- Deadline sensitivity
In task Task3.default_profile, the deadline for response Stimulus3.Response3_1 can be met for execution time up to 4000 cycles (500 us).
In task Task3.default_profile, the deadline for response Stimulus3.Response3_2 can be met for execution time up to 4000 cycles (500 us).
In task Task2.default_profile, the deadline for response Stimulus2.Response2 can be met for execution time up to 10000 cycles (1.25 ms).
--- System sensitivity to execution and lock times
In task Task4.default_profile, the system can be schedulable for execution time up to 39176 cycles (4.897 ms).
- resource Resource1 can be locked for up to 8045 cycles (1.005625 ms).
In task Task5.default_profile, the system can be schedulable for execution time up to 39176 cycles (4.897 ms).
- interrupt level 1 can be locked for up to 7748 cycles (968.5 us).
In task Task1.default_profile, the system can be schedulable for execution time up to 17088 cycles (2.136 ms).
- resource Resource1 can be locked for up to 8045 cycles (1.005625 ms).
- interrupt level 1 can be locked for up to 7748 cycles (968.5 us).
In task Task3.default_profile, the system can be schedulable for execution time up to 21038 cycles (2.62975 ms).
In task Task2.default_profile, the system can be schedulable for execution time up to 19342 cycles (2.41775 ms).
- resource Resource1 can be locked for up to 19342 cycles (2.41775 ms).
In interrupt Bursting.default_profile, the system can be schedulable for execution time up to 5195 cycles (649.375 us).
In interrupt Timer1.default_profile, the system can be schedulable for execution time up to 2517 cycles (314.625 us).
In interrupt Timer2.default_profile, the system can be schedulable for execution time up to 4771 cycles (596.375 us).
--- System sensitivity to clock speed
The system remains schedulable if processor clock speed is reduced to 69.81% of its current value.

Figure 15:12 - Viewing Sensitivity Analysis Results in Text Format

The sensitivity analysis results can be used to modify your application. The following sections explain how you can interpret the results.

15.3.1 Sensitivity to Clock Speed

The current speed that is displayed will always be 100% of the CPU clock frequency. The new speed will be a percentage of the clock speed required so that the system is schedulable.

If the new figure is less than 100% then there is scope for reducing the clock speed of your target hardware. This can be useful, for instance, in the case of devices that must minimize power consumption.

If the new figure is greater than 100% then you will have to increase your CPU clock speed to make the system schedulable. The analysis gives you the smallest increase required for your application to become schedulable.

15.3.2 Sensitivity to Execution Times

When the sensitivity analysis finishes, the workspace shows sensitivity to execution times by default. Each bar shows the current execution time in blue and the maximum execution time for that task or ISR in green.

If the system is not schedulable, then the bar will show a red overrun. This indicates the amount by which the current task or ISR is exceeding its maximum permitted execution time, such that the system is schedulable. Overrun can be seen in Figure 15:13.

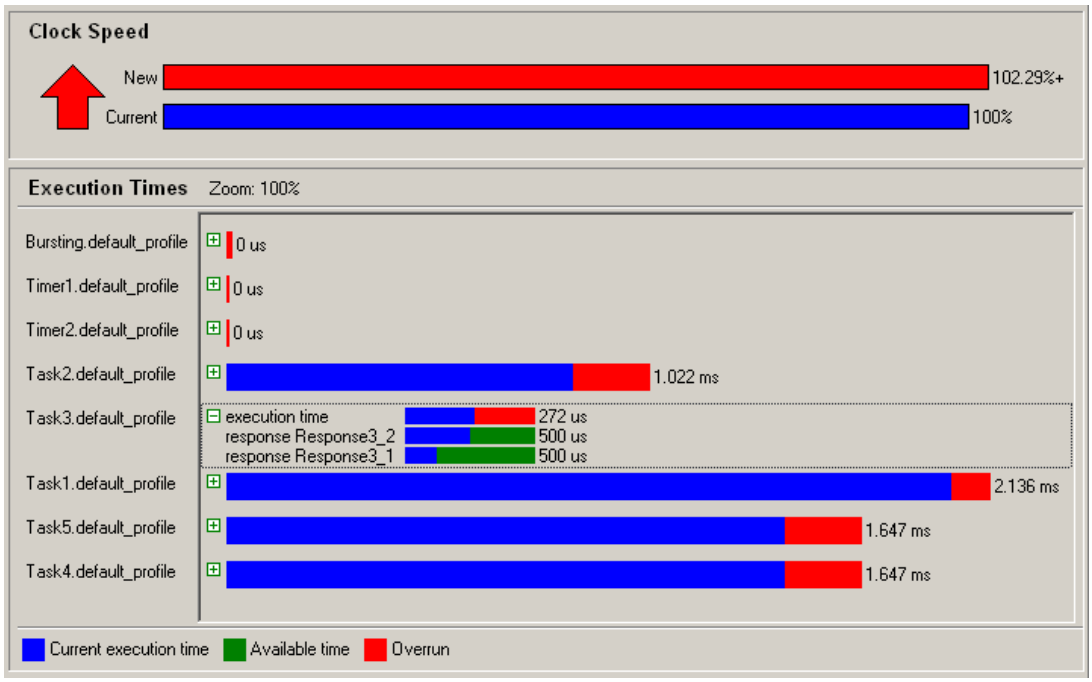


Figure 15:13 - Sensitivity Analysis Showing Overrun

The reported limits of maximum execution are mutually exclusive. For any single task or ISR you can change the execution time to the value shown by sensitivity analysis and the system will remain (or become) schedulable.

If you change the execution time of more than one task or ISR you will have to re-run sensitivity analysis to validate those changes.

In some cases the analysis will work out that changes to some of your tasks or ISRs will have no effect (even if they execute in zero time). These will not be shown graphically. The textual report will state, however, that it will have no effect if you change the execution times for those task and ISRs.

This feature allows you to target your effort. It is extremely useful when you are trying to reduce execution times to make an unschedulable system schedulable.

When sensitivity analysis finishes, the workspace shows sensitivity to execution times by default. You can view sensitivity to resource and interrupt lock times by expanding the information for each task or ISR, shown in Figure 15:14.

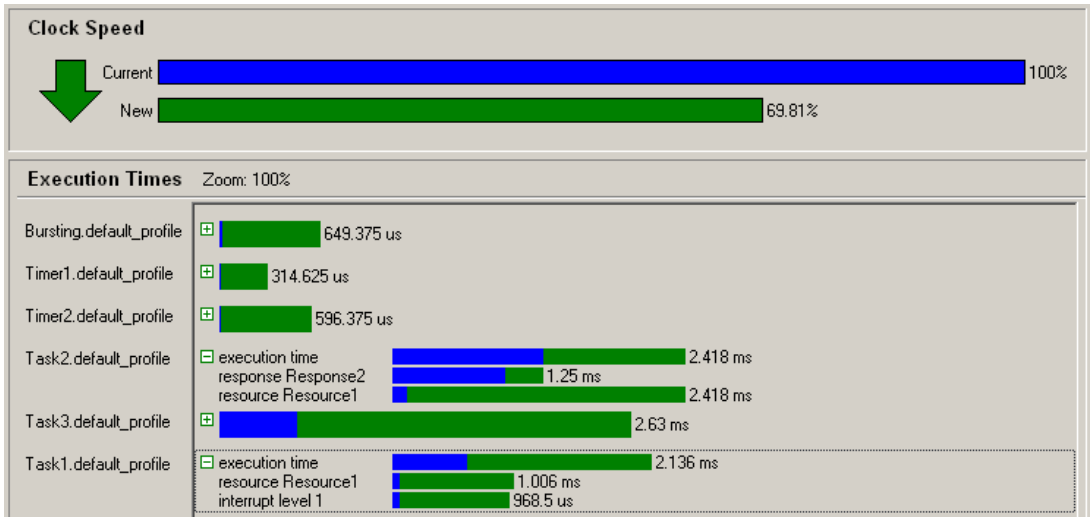


Figure 15:14 - Viewing Interrupt and Lock Times from Sensitivity Analysis

For each resource or interrupt that is locked or disabled by the task or ISR, sensitivity analysis reports the current lock time and the maximum time for which the resource can be locked or the interrupt disabled.

Resource and interrupt lock times for each task or ISR are mutually exclusive. If there is additional overhead for resource and interrupt locking times, you can only use the maximum execution time determined by sensitivity analysis for a *single* resource or interrupt lock. You can, of course, use part of the time for each lock and re-run sensitivity analysis to validate the changes.

15.3.3 Sensitivity to Deadlines

When the sensitivity analysis finishes, the workspace shows sensitivity to execution times by default. You can view sensitivity to explicit response deadlines by expanding the information for each task or ISR.

For each response with a specified deadline, sensitivity analysis reports the current execution time of the response implementation (this is specified by you when constructing the timing model) and the maximum amount of time for which the response implementation can execute, while the system remains schedulable. Any overruns are shown in red.

The results of sensitivity analysis to deadlines for each task or ISR are complementary. You can set the execution time for each response to the maximum value determined by analysis and the system will remain schedulable.

15.4 Best Task Priorities Analysis

Best task priorities analysis is used to allocate task priorities to make the system schedulable, if this is possible. It will also determine the tasks that could share an internal resource to minimize the stack space required by your application. If your application already includes internal resources, they are included in the analysis.

The RTA-OSEK GUI presents the results of the analysis graphically. An example is shown in Figure 15:15.

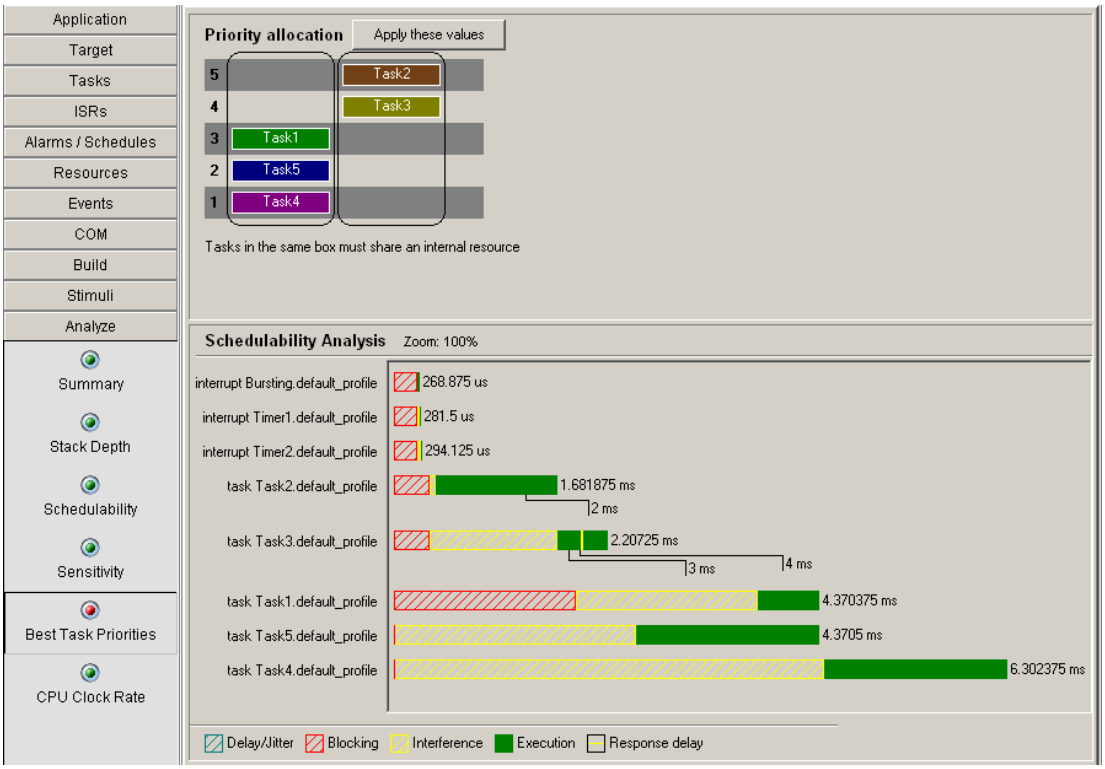


Figure 15:15 - Viewing Best Task Priority Analysis Results in Graphical Format

The result of Best task priorities analysis is also available in textual form as shown in Figure 15:16.

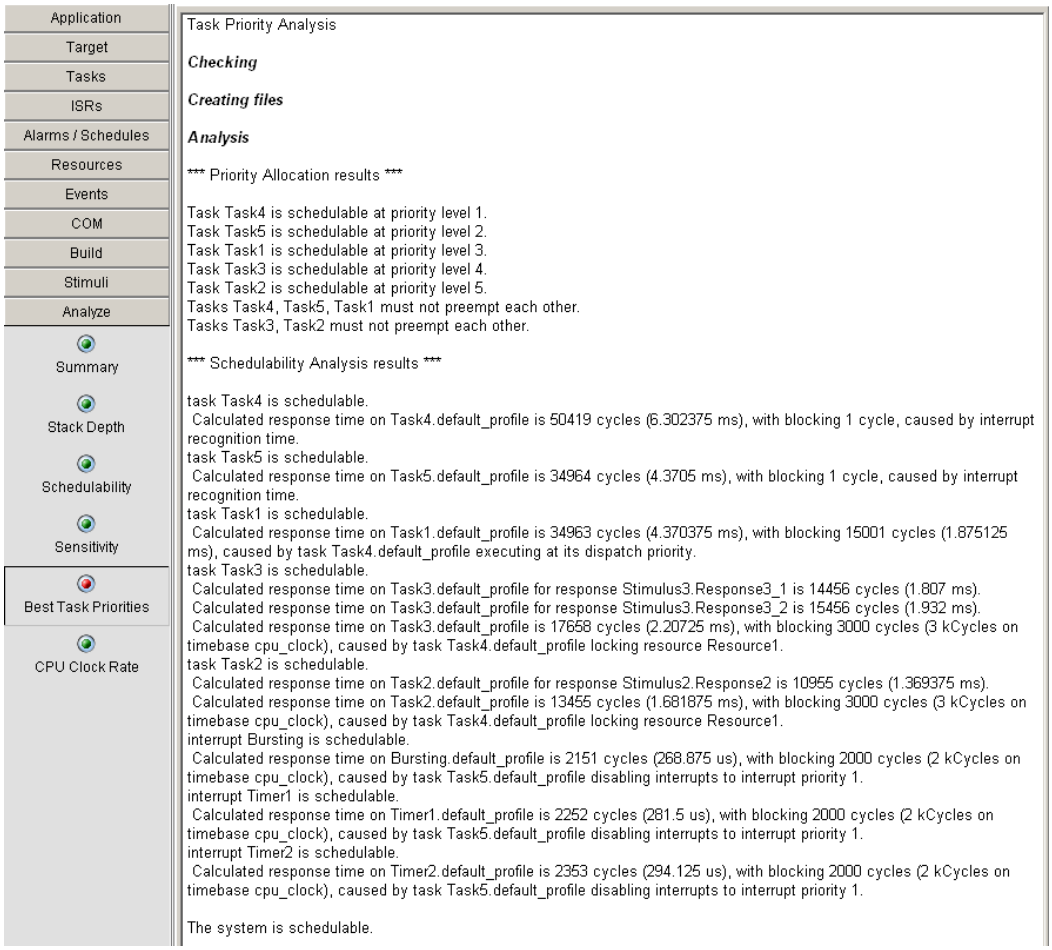

Application
Target
Tasks
ISRs
Alarms / Schedules
Resources
Events
COM
Build
Stimuli
Analyze
Summary
Stack Depth
Schedulability
Sensitivity
Best Task Priorities
CPU Clock Rate
Task Priority Analysis
Checking
Creating files
Analysis
*** Priority Allocation results ***
Task Task4 is schedulable at priority level 1.
Task Task5 is schedulable at priority level 2.
Task Task1 is schedulable at priority level 3.
Task Task3 is schedulable at priority level 4.
Task Task2 is schedulable at priority level 5.
Tasks Task4, Task5, Task1 must not preempt each other.
Tasks Task3, Task2 must not preempt each other.
*** Schedulability Analysis results ***
task Task4 is schedulable.
Calculated response time on Task4.default_profile is 50419 cycles (6.302375 ms), with blocking 1 cycle, caused by interrupt recognition time.
task Task5 is schedulable.
Calculated response time on Task5.default_profile is 34964 cycles (4.3705 ms), with blocking 1 cycle, caused by interrupt recognition time.
task Task1 is schedulable.
Calculated response time on Task1.default_profile is 34963 cycles (4.370375 ms), with blocking 15001 cycles (1.875125 ms), caused by task Task4.default_profile executing at its dispatch priority.
task Task3 is schedulable.
Calculated response time on Task3.default_profile for response Stimulus3.Response3_1 is 14456 cycles (1.807 ms).
Calculated response time on Task3.default_profile for response Stimulus3.Response3_2 is 15456 cycles (1.932 ms).
Calculated response time on Task3.default_profile is 17658 cycles (2.20725 ms), with blocking 3000 cycles (3 kCycles on timebase cpu_clock), caused by task Task4.default_profile locking resource Resource1.
task Task2 is schedulable.
Calculated response time on Task2.default_profile for response Stimulus2.Response2 is 10955 cycles (1.369375 ms).
Calculated response time on Task2.default_profile is 13455 cycles (1.681875 ms), with blocking 3000 cycles (3 kCycles on timebase cpu_clock), caused by task Task4.default_profile locking resource Resource1.
interrupt Bursting is schedulable.
Calculated response time on Bursting.default_profile is 2151 cycles (268.875 us), with blocking 2000 cycles (2 kCycles on timebase cpu_clock), caused by task Task5.default_profile disabling interrupts to interrupt priority 1.
interrupt Timer1 is schedulable.
Calculated response time on Timer1.default_profile is 2252 cycles (281.5 us), with blocking 2000 cycles (2 kCycles on timebase cpu_clock), caused by task Task5.default_profile disabling interrupts to interrupt priority 1.
interrupt Timer2 is schedulable.
Calculated response time on Timer2.default_profile is 2353 cycles (294.125 us), with blocking 2000 cycles (2 kCycles on timebase cpu_clock), caused by task Task5.default_profile disabling interrupts to interrupt priority 1.
The system is schedulable.

Figure 15:16 - Viewing Best Task Priorities Analysis Results in Text Format

In priority allocation, each task is shown with its calculated best priority. Tasks are also grouped according to whether they could share an internal resource (and therefore minimize application stack space). The lower section of the workspace shows the schedulability analysis assuming that you were to apply these changes.

Maximizing the number of tasks that share an internal resource has two important effects on the system:

- Total required stack for the system is minimized. The worst-case stack usage for an arbitrary set of tasks is normally the sum of the worst-case stack usages for each of the tasks. When these tasks share an internal resource, the worst-case stack usage is the single largest stack usage of *any* of the tasks sharing the resource.
- A system is expected to become less schedulable as tasks share an internal resource (because slack time is traded with reduced stack usage). However, if the additional overhead of switching from one task to another is a significant proportion of the task execution time, then sharing an internal resource between these tasks may improve schedulability.

15.4.1 Required Lower Priority Tasks

If you know that certain tasks must have a higher priority than others, you should enter these constraints into the RTA-OSEK GUI.

You can do this, for example, to ensure a specific execution order. So, for instance, if `Task3` and `Task4` may be activated together, but `Task3` prepares some data that is used by `Task4`, then `Task3` must execute first. `Task3` is, therefore, a required lower priority task for `Task4`.

You should specify the required lower priority tasks for the tasks that are of interest. This information has been specified in Figure 15:17.

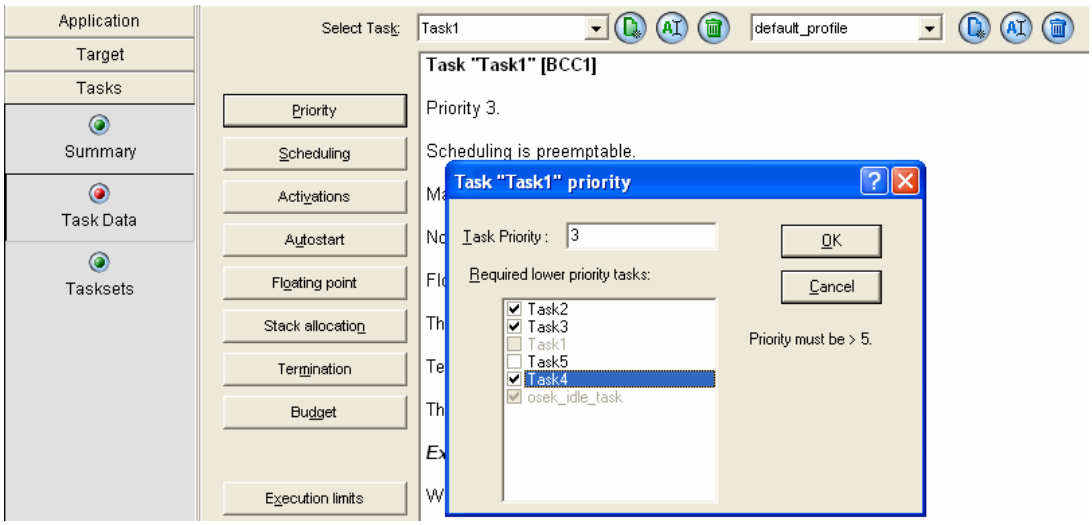


Figure 15:17 - Selecting the Required Lower Priority Tasks

When best task priorities analysis performs priority allocation, it uses the required lower priority constraints and searches for the priority ordering that best allows the system to meet its schedulability requirements.

In general, when using automatic priority allocation, the fewer priority constraints that are placed on the system, the better the priority ordering that can be defined. This means that only priority constraints that are absolutely necessary should be given in the priority constraints declaration.

15.5 CPU Clock Rate Optimization

CPU clock rate optimization is similar in concept to best task priorities, but it optimizes for time rather than space. It looks for the lowest possible clock rate that gives a schedulable system.

CPU clock rate optimization will rearrange task priorities if this results in a system that is schedulable at a lower clock frequency.

Figure 15:18 shows the results of CPU clock rate optimization in text format.

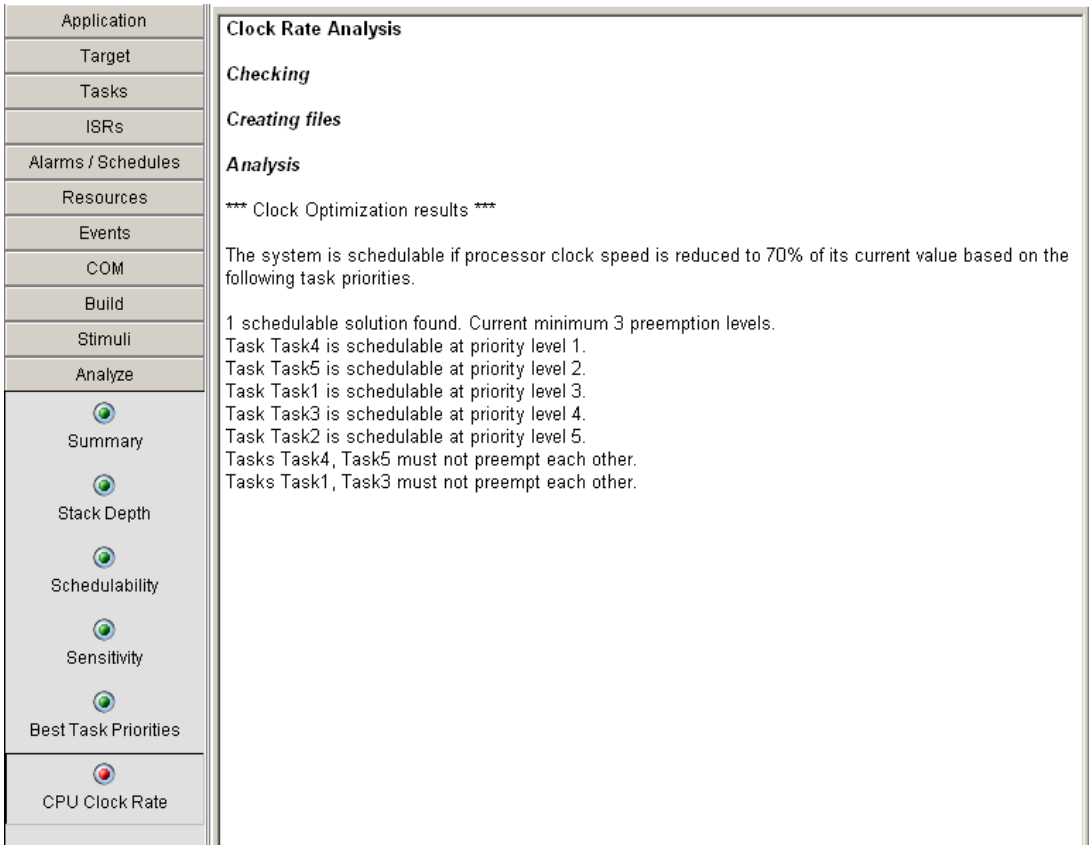

Application
Target
Tasks
ISRs
Alarms / Schedules
Resources
Events
COM
Build
Stimuli
Analyze
Summary
Stack Depth
Schedulability
Sensitivity
Best Task Priorities
CPU Clock Rate
Clock Rate Analysis
Checking
Creating files
Analysis
*** Clock Optimization results ***
The system is schedulable if processor clock speed is reduced to 70% of its current value based on the following task priorities.
1 schedulable solution found. Current minimum 3 preemption levels.
Task Task4 is schedulable at priority level 1.
Task Task5 is schedulable at priority level 2.
Task Task1 is schedulable at priority level 3.
Task Task3 is schedulable at priority level 4.
Task Task2 is schedulable at priority level 5.
Tasks Task4, Task5 must not preempt each other.
Tasks Task1, Task3 must not preempt each other.

Figure 15:18 - Viewing CPU Clock Rate Optimization Analysis Results in Text Format

Figure 15:19 shows the results of CPU clock rate optimization in graphical format.

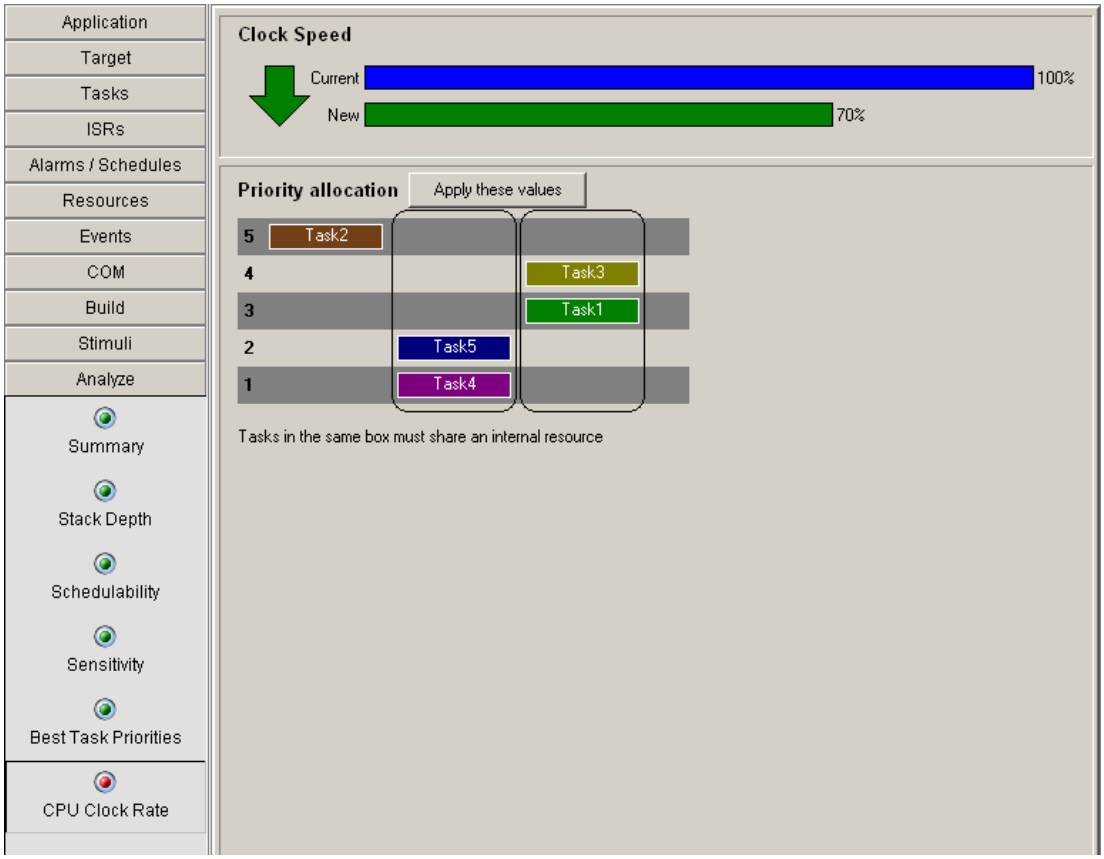


Figure 15:19 - Viewing CPU Clock Rate Optimization Analysis Results in Graphical Format

Clock optimization can be thought of as a combination of sensitivity analysis and priority allocation from best task priorities analysis.

- As with clock sensitivity (explained in Section 15.3), it is assumed that the effect of changing the clock frequency only impacts on execution times (including critical execution times, resource and interrupt lock times). Deadlines and delays between alarms and arrivalpoints on schedules are not scaled.
- As with priority allocation, task priorities may be rearranged. You should specify required lower priority tasks where there are necessary constraints on reprioritization.

If it is important that your system has critical requirements for power consumption or heat dissipation, then you should consider using clock optimization on your final application.

Performing Analysis Summary

- RTA-OSEK provides facilities for analyzing the timing model of your application using RTA-OSEK Planner.
- Stack analysis allows you to determine the worst-case stack usage for your application, accounting for situations where stack space can be effectively overlaid due to the calculated run-time behavior of your application.

- Schedulability analysis tells you whether or not every response deadline in your application will be met at run-time for all possible arrivals of stimuli. If your application is found to be unschedulable, there are a number of approaches your can use to make it schedulable.
- Sensitivity analysis lets you explore the boundaries of your application, either to detect areas that are making your system unschedulable or to look at the scope for possible future enhancements.
- Best task priorities analysis determines the best priority allocation for your tasks, such that the system is schedulable. Required lower priority tasks can be specified for tasks whose execution ordering is important. Best task priorities analysis also determines which tasks can share an internal resource, so that stack space can be minimized.
- CPU clock rate optimization is similar to best task priorities, but optimizes for minimum CPU clock rate rather than for minimum stack space.

16 Using RTA-OSEK from the Command Line

16.1 Overview of Operation

16.1.1 Functionality

The tool `rtabuild` is a command line program that invokes the various tools of the RTA-OSEK development suite to provide the following functions:

- **Build Kernel and application support data.**
  This is used to create the RTA-OSEK header, C and assembler files that you need in order to compile and link your application.
- **Schedulability analysis.**
  Determines whether all tasks and ISRs meet their deadlines and other constraints in the worst-case.
- **Sensitivity analysis.**
  Calculates how far a range of timing parameters for each task and ISR can be varied to achieve a system that is only just schedulable.
- **Best Task Priorities.**
  Allocates task priorities to give the minimum number of preemption levels, commensurate with a schedulable system.
- **Clock Optimization**.
  Determines the lowest processor clock rate that can be achieved for a schedulable system and the task priorities that are necessary to run at this rate.

`rtabuild` reads input configuration file(s) written in OIL syntax.

16.1.2 Messages

During its operation, `rtabuild` reports various messages. They can be:

- **Information** messages.
  Reports useful information such as the amount of memory used or the size of a data structure.
- **Warning** messages.
  Occur when the input file specifies an unusual condition, something that is redundant or a value that cannot be represented precisely and is therefore subject to rounding error. When warnings have been produced, the output files are created and the application can be built.
- **Error** messages.
  Generated where there are conflicts in the input file that make it impossible to perform analysis correctly or produce correct output. The tool attempts to report all the errors it can find and then exits. All errors should be removed and the operation should be repeated before attempting to use the results or output of `rtabuild`.
- **Fatal** messages.
  Caused where there are conditions in the input file, from which recovery

is not possible. `rtabuild` stops processing immediately after detecting and reporting a fatal error message.

16.1.3 Return Values

At the end of its execution `rtabuild` will return a code as indicated by the table below:

Value	Description
0	Success in the requested operation.
1	Termination as a result of an error or fatal message.
2	User cancelled processing (`ESC`) while running RTA-OSEK Planner.
3	User abort (`^C/^Break`).
4	System is not schedulable. (`0` is returned if `-u` command line option is specified).
5	Priority allocation failed to generate a schedulable system.

16.1.4 Command Line Options

`rtabuild` is invoked from the command line using:

```
rtabuild [[*options*]] [*input_file*]
```

Command line options are identified by a preceding hyphen. Any valid combination of command line options can be specified in any order.

One or more input files can be specified on the command line. All files must be written using OIL syntax. Where there is more than one input file specified, the files are processed in the order that they appear. The text of each file is appended to the previous file to create a temporary file that is processed.

The available options are summarized in the following table. Note that if none of the options `-a`, `-p` or `-s` are selected, `rtabuild` automatically creates the kernel and application support files.

Options	Description
General options:	
-e	Treat warnings as errors. If warnings have been output, rtabuild exits preceded by error E0402.
-i<path>	Add to the search path for include files.
-k	Keep intermediate files. During execution, rtabuild may create files that contain information that is passed to other command-line tools. These files are normally deleted after use.
-o[name]	Generate listing file. If name is not specified it is constructed from the root part of the first input file with a .lst extension. The listing file contains all output from the launched programs, including commentary, such as the command lines invoked.
-v	Turn on verbose information detailing progress.
-w[vvvv]	Ignore all warnings (-w) or ignore a specific numbered warning (-wvvvv). If all warnings are ignored, no warnings are shown on screen or in the listing file. If a specific warning vvvv is to be ignored, the vvvv characters are used to match the warning message code. If a matching message occurs, the warning is removed from screen and listing file output. Warnings are ignored before the -e option can have an effect.
Build options:	
-d<status>	Override the build status specified in the input OIL file. <status> = s for STANDARD <status> = t for TIMING <status> = e for EXTENDED
-g	Suppress generation of vector table. This is target specific. Only relevant when generating output files.
-l	Allow very long schedules. The limit of 64K bytes for the size of target schedules is turned off. Only relevant when generating output files.
-xname	Save the input data in a new output OIL file called name. This has the same effect as saving a file in the RTA-OSEK GUI. If the -d option is used, the file generated will reflect the value specified.

Options	Description
-yname	Extract the RTA-OSEK custom build script into a file called name. Usually this is a batch file that can be run from the command-line to achieve the same effect as custom build in the RTA-OSEK GUI. Also causes rtkbuild.bat to be generated.
Analysis options:	
-a[n]	Selects schedulability analysis and sets the schedulability algorithm used. n determines the depth of the analysis. It may take values 1 to 9 (1 is tractable analysis and 9 is exact analysis and the other values are reserved for future use). Note that -a is equivalent to -a1. Specifying this option with -s or -p, sets the schedulability algorithm used for sensitivity analysis and priority allocation respectively.
-c[n]	Perform clock optimization to determine the lowest clock rate at which a schedulable system can be achieved. Task priorities may be adjusted. The priority n determines the 'packing aggressiveness' for the priority allocation. It defaults to 1 if not specified. This option cannot be selected if either -p or -s is also selected
-nname	Ignore named task/interrupt/profile during timing analysis.
-p[n]	Select automatic task priority allocation. n determines the 'packing aggressiveness'. n defaults to 1 if not specified. This option cannot be selected if either -c or -s is also selected.
-s[name]	Perform sensitivity analysis. If name is specified, perform sensitivity analysis for the task or interrupt with that name. Otherwise perform analysis for each executable object in turn. This option cannot be selected if either -c or -p is also selected.
-u	Do not treat non-schedulability as an error. Only relevant for analysis.

16.1.5 Output Files

During processing, rtabuild may generate intermediate, temporary and listing files. Temporary files are always removed on completion. Intermediate files are normally deleted, but can be retained using the -k option. Listing files are only generated if the -o option is selected.

17 Using RTA-OSEK with RTA-TRACE

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably enhanced application, it provides the embedded application developer with a unique set of services to assist in debugging and testing a system. Foremost amongst these is the ability to see exactly what is happening in a system at runtime with a production build of the application software.

Configuration of RTA-TRACE parameters for RTA-OSEK is carried out using the RTA-OSEK GUI. The GUI is largely self-explanatory, so this section will simply describe a set of tasks and how one might achieve them.

It is assumed that you have some knowledge of using the RTA-OSEK configuration tool, so creation/configuration of the application is not discussed here.

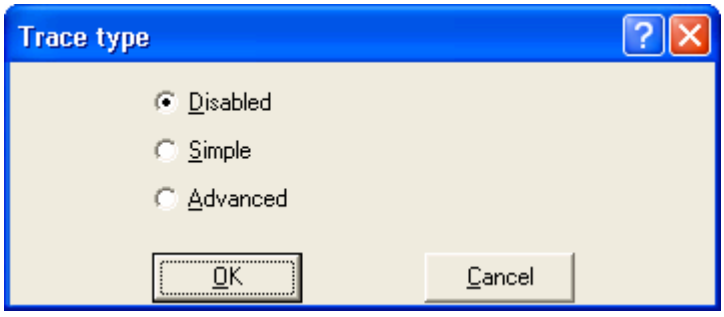
All of the configuration tasks relating to RTA-TRACE are accessed from the **RTA-TRACE** tab at the bottom left-hand side of the GUI.

17.1 Configuration

The following options can be set from this pane:

Trace Type Disables or enables (either simple or advanced) tracing.

Advanced tracing provides more detailed tracing at than simple tracing, with a corresponding increase in trace-records.



Compact IDs The compact trace format saves buffer space by only allowing 4 bits for task tracepoint ID values, and 8-bits for tracepoint and interval ID values. Other identifiers (Tasks, Resources etc.) use 8 bits.

If compact identifiers are not selected, 12 bits are used for tracepoint, task tracepoint, and interval ID values, with 16 bits being used for other identifiers.

Compact Time Select *compact* (16-bit); or *extended* (32-bit) time format. This option may not be available for every target.

Trace Stack Select whether to record stack usage or not

Target Triggering Select whether runtime target triggering is available

Buffer Size

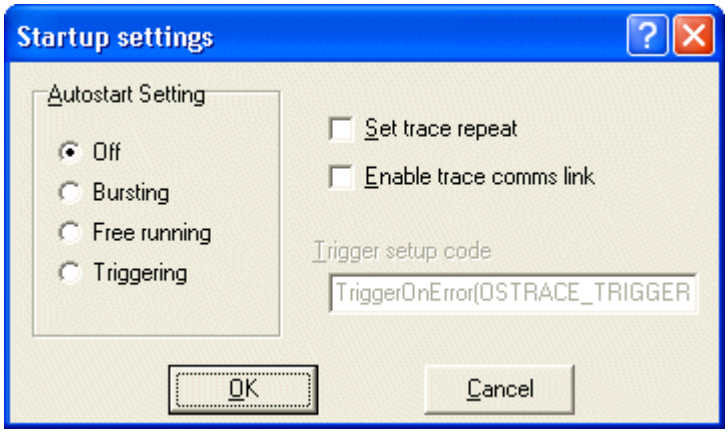
This controls the size of the buffer reserved on the target for the tracing information. Note that the number is in records, not bytes, so the actual buffer size in bytes depends upon sizes selected for time and identifier.



Autostart

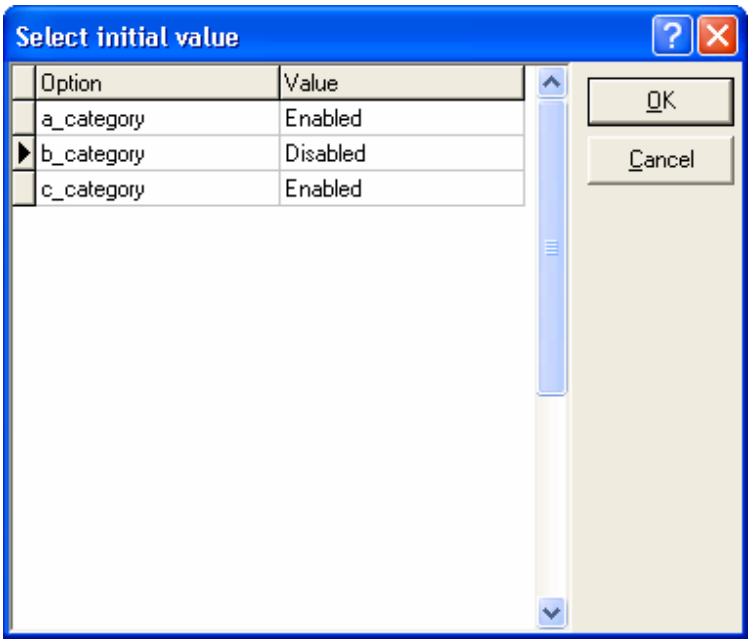
Select whether tracing is started automatically, and which trace mode to start in.

For triggering operation, the trigger setup (`TriggerOn...`) code is entered here. Details of the triggering API can be found in the *RTA-TRACE User Manual*.



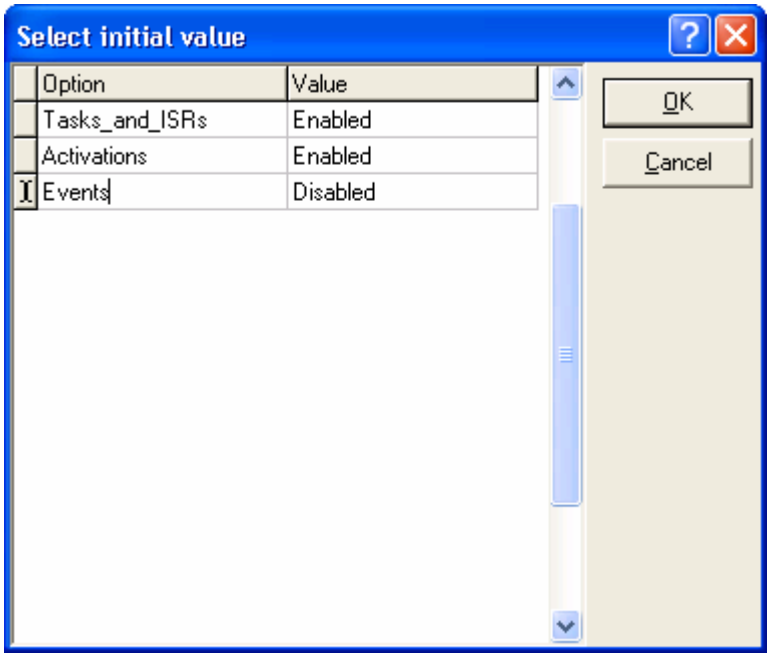
Initial Categories

If run-time categories have been defined (See the *RTA-TRACE User Manual* for more details about categories), this dialog allows you to choose which run-time, user-defined categories are enabled when tracing starts. Below, we can see three run-time, user-defined categories, one of which is initially disabled.

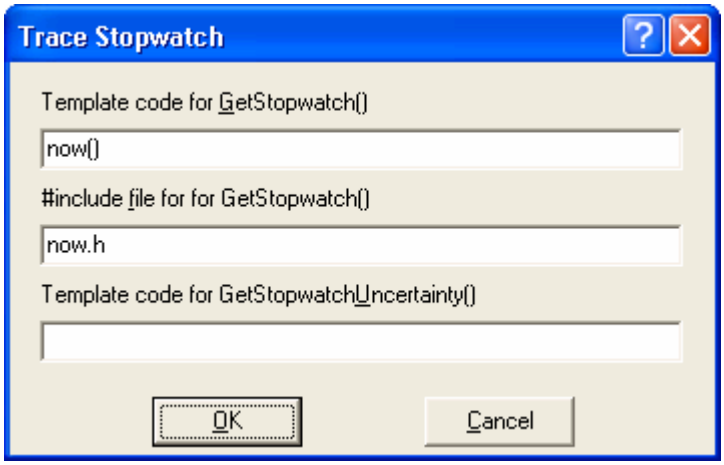


Initial Classes

Choose which record classes are enabled when tracing starts. Below we can see that task and ISR, activation, and event tracing are able to be enabled and disabled at run-time and that, initially, event tracing is disabled.

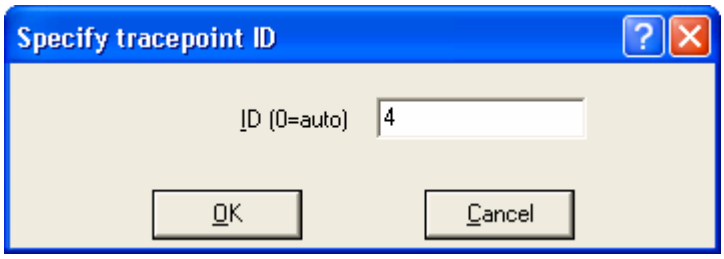


Stopwatch This dialog allows the user to specify what function is used to implement `GetStopwatch()`. In the dialog below, this is a user-supplied function called `now()`. The header file supporting this function is called `now.h`.

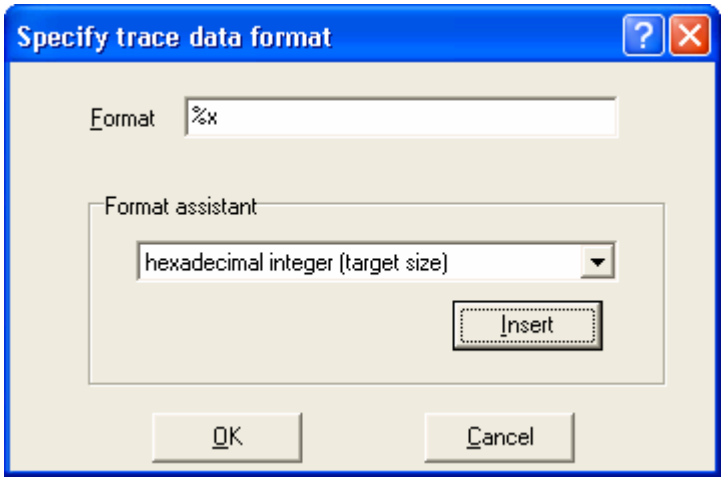


17.2 Tracepoints

This pane allows tracepoints to be defined. New tracepoints are initially given auto-generated identifiers, but this can be over-ridden using the **ID** button:



If the tracepoint has associated data, it is possible to supply a *format-string* (see section 17.8 for more information about format strings) to govern how the data will be displayed:



17.3 Task Tracepoints

This pane allows task-tracepoints to be defined. New task-tracepoints are initially given auto-generated identifiers, but this can be over-ridden using the **ID** button as for tracepoints.

Format strings are entered in the same way as for tracepoints.

17.4 Intervals

This pane allows intervals to be defined. New intervals are initially given auto-generated identifiers, but this can be over-ridden using the **ID** button as for tracepoints.

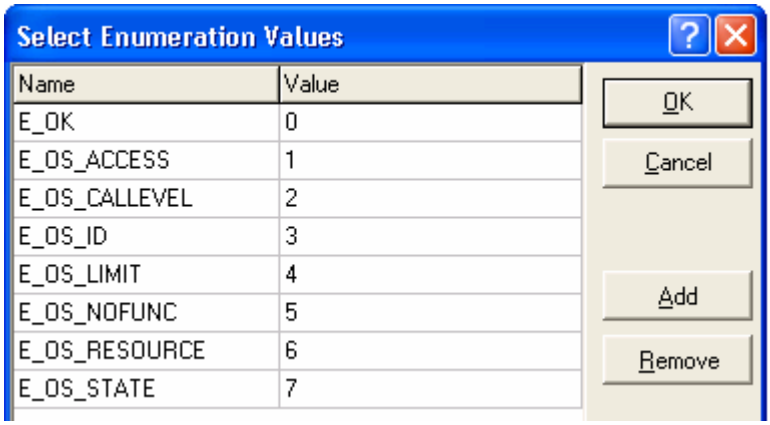
Format strings are entered in the same way as for tracepoints.

17.5 Categories

This pane allows trace categories to be defined, along with their mask-value. See the *RTA-TRACE User Manual* for more details about categories. Categories can be always enabled, always disabled, or enabled/disabled at run-time by using the **filter** pane (see 17.7 below).

17.6 Enumerations

This pane allows enumerated identifiers to be specified, along with numeric values. The example shown illustrates how the first few OSEK error codes might be enumerated.



17.7 Filters

This pane configures the filtering of event classes and categories. Events can be either always enabled; always disabled; or enabled/disabled at run-time. Run-time classes can be initially enabled or disabled based upon the 'Initial Classes' button on the configuration pane (See section 17.1).

17.8 Format Strings

Format strings specify how a tracing item's data should be displayed. Simple numeric data can be displayed using a single format specifier. More complex data, e.g. a C `struct`, can be displayed by repeatedly moving a cursor around the data block and emitting data according to more complex format specifiers.

If a format string is not supplied, data is displayed in the following manner:

- if the data size is no greater than the size of the target's `int` type, data is decoded as if "%d" had been specified.
- Otherwise the data is displayed in a hex dump, e.g.

```
0000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

- A maximum of 256 bytes is shown.

Note: when format specifiers are given, the target's endian-ness is taken into account. When a hex dump is shown, the target's memory is dumped byte-for-byte. In particular, you may not get the same output from a hex dump as from the `%x` format specifier.

17.8.1 Rules

Format strings are similar to the first parameter to the C function `printf()`:

- Format strings are surrounded by double-quote (`"`) symbols.
- A format string may contain two types of object: ordinary characters, which are copied to the output stream, and format elements, each of which causes conversion and printing of data supplied with the event.
- A format element comprises a percent sign, zero or more digits and a single non-digit character, with the exception of the `%E` element – see below.
- The format element is decoded according to the rules in the table below, and the resulting text is added to the output string.
- The special format element `%%` emits a `%`.
- In addition to ordinary characters and conversion specifications, certain characters may be emitted by using a 'backslash-escape-sequence'. To emit a double-quote (`"`) character, `\"` is used, and to emit a `\` character, `\\` is used.
- The optional size parameter to integer format specifiers defines the field's width in bytes. Valid values are 1, 2, 4 or 8.

Note: An important difference from `printf()` is that the cursor does not automatically move on from the current field when a field is emitted. This is to facilitate multi-format output of a single field.

Format Element	Meaning
`%offset@`	Moves the cursor *offset* bytes into the data. This can be used to extract values from multiple fields in a structure.
`%[size]d`	Interpret the current item as a signed integer. Output the value as signed decimal.
`%[size]u`	Interpret the current item as an unsigned integer. Output the value as unsigned decimal.
`%[size]x`	Interpret the current item as unsigned integer. Output the value as unsigned hexadecimal.
`%[size]b`	Interpret the current item as an unsigned integer. Output the value as unsigned binary.
`%enum[:size]E`	Interpret the current item as an index into the enumeration class who's ID is *enum*. Emit the text in that enumeration class that corresponds with the item's value. The enumeration class should be defined using `ENUM` directives. An exception is implicitly defined enum class 99, which is the set of ERCOS^{EK} errors.
`%F`	Treat the current item as an IEEE 'double'. Output the value as a double, in exponent format if necessary.
`%?`	Emit in the form of a hex dump.
`%%`	No conversion is carried out; emit a %

17.8.2 Examples

Description	Format String	Example	Notes
A native integer displayed in decimal and hexadecimal.	`"%d 0x%x"`	`10 0xA`	The `"0x"` is not emitted by the `%x` format specifier but is specified in literal characters in the string. Absence of size specifier means the target's `int` size is assumed. This example is a 16-bit processor.
A single unsigned byte representing a percentage.	`"%1u%%"`	`73%`	Use of size specifier of 1 byte. Use of `%%` to emit `%`.

`struct{`
`int x;`
`int y;`
`};`
… On a 32-bit processor.
	`"(%d,%4@%d)"`	`(20,-15)`	Use of `%offset@` to move to byte-offset within the structure.

A value of type `enum e_Rainbow`, (defined as the colours of the rainbow!)	`"%1E"`	`Yellow`	The number `1` refers the ID of the enum class in the `ENUM` directives, not to the width of the field.

18 Index

A

a.out 12-9
Absolute periodic alarms 10-15
Accessors
  About 8-4
  Creating 8-5, 8-6
  Receiving messages 8-8
  Renaming 8-6
  Sending messages 8-8
  Transmission mechanisms ... 8-6
  WithCopy 8-6, 8-12
  WithoutCopy 8-6, 8-12
Actions
  Alarms 10-4
  Events 10-10
ActivateTask() .4-12, 4-17, 4-21, 13-14
Activating
  Tasks 4-11
Activation
  Advanced task 4-17
  Auto-activation 4-19
  Buffered 15-9
  By a message 4-18
  By a periodic schedule 4-18
  By a planned schedule 4-18
  By an alarm 4-18
  By an event 4-18
  Counts 2-3, 4-11, 15-7
  Direct 4-17, 4-21
  Direct activation chains 4-21
  Direct tasks 4-17, 4-21
  Explicit 4-3
  Fast 4-15
  Indirect 14-27
  Non-queued 15-8
  Tasks 4-11
  Upward 14-4
Advanced
  Error logging 13-7
Advanced concepts
  Counters and alarms 10-9
  Error handling & execution monitoring 13-6
  Events 7-5
  Interrupts 5-9
  Messages 8-11
  Modeling the application for the RTA-OSEK Planner 14-19
  Resources 6-8
  Schedules 11-12
  Startup and shutdown 12-10
  Tasks 4-13
Advanced schedules 11-13
  About 11-3
  Driver callbacks 11-15, 11-16
Advanced task activation
  About 4-17
  Activation by alarm 4-18
  Activation by event 4-18
  Activation by message 4-18
  Activation by periodic schedule 4-18
  Activation by planned schedule 4-18
  Auto-activation 4-19
  Direct activation 4-17, 4-21
Alarms
  About 10-1, 10-16
  Absolute **10-1**, 10-8, 10-13, 10-14
  Absolute periodic 10-15
  Actions 10-4
  Activating tasks 4-18
  Analysis 14-19
  Aperiodic 10-15
  Autostarted 10-13
  Autostarting 10-5, 12-12
  Callbacks 10-9
  Canceling 10-9
  Cyclic .. **10-1**, 10-6, 10-8, 10-13
  Declaring 10-2
  Expiry **10-1**, 10-13
  Relative **10-1**, 10-7, 10-14
  Setting 10-6

Setting events 7-6
Single-shot .. **10-1**, 10-6, 10-15
Synchronization... 10-14, 12-13, 14-19
Analysis
Accuracy 14-3
Alarms 14-19
Best task priorities . 15-1, 15-16, 15-21
Configuring applications for 14-4
CPU clock rate. 15-1, 15-19, 15-21
Implicit deadlines..... 14-7
Options 15-1, 15-21
Overrides..... **14-27**
Performing 15-1, 15-21
Pessimism.... 14-3, 14-14, 14-19
Requirements 14-3
Schedulability . 3-57, 15-1, 15-4, 15-21
Sensitivity 3-59, 15-1, 15-12, 15-21
Stack depth 15-1, 15-21
Tractable 15-12
Aperiodic alarms..... 10-15
API calls
ActivateTask() . 4-12, 4-17, 4-21, 13-14
ChainTask() ... 4-8, 4-18, 4-25
ChainTaskset()..... 4-8, 4-25
ClearEvent()..... 7-4
CloseCOM() 8-10
DisableAllInterrupts() 5-8
EnableAllInterrupts()..... 5-8
GetAlarm() 10-8, 10-13
GetAlarmBase() 10-11
GetArrivalpointDelay() 11-20
GetArrivalpointTasksetRef() . 11-21
GetExecutionTime() 13-11
GetResource()6-2, 6-3, 6-4, 6-5, 6-8, 6-10
GetStackOffset().. 13-12, 13-13, 13-14
GetStopwatch() 3-7, 3-27, 13-9, 13-10, 13-11, 15-3, 15-4
GetStopwatchUncertainty() . 13-9, 13-10
InitCOM() 8-10
OverrunHook()..... 13-2, 13-10
ReadFlag() 8-14
ReleaseResource() . 6-2, 6-3, 6-4, 6-8, 6-10
ResetFlag()..... 8-14
ResumeAllInterrupts() 5-8, 8-14, 10-9
ResumeOSInterrupts()..... 5-8
Schedule()3-54, 4-14, 4-15, 6-7, 14-4
SendMessage() 13-8
SetAbsAlarm() . 10-9, 10-13, 10-14
SetArrivalpointDelay() 11-20
SetEvent() 7-4, 7-6, 10-10
SetRelAlarm()..... 10-9, 10-13
SetScheduleNext(). 11-13, 11-21
ShutdownOS() 3-20, 3-21, 3-49, 12-10, 12-13, 13-3
StackFaultHook()13-2, 13-5, 13-6
StartCOM()..... 8-10, 8-11
StartOS()3-20, 3-21, 3-49, 4-12, **4-19**, 5-8, 8-10, 10-5, 10-13, 11-12, 12-9, 12-10, 12-11, 12-12, 12-13, 13-14
StartSchedule() 11-10, 11-11
StopCOM() 8-10
StopSchedule() 11-12
SuspendAllInterrupts() 5-8, 10-9
SuspendOSInterrupts()..... 5-8
TerminateTask() ... 4-8, 4-23, 4-25, 6-9
Tick_<CounterID>() 10-5
TickSchedule_<ScheduleID> 11-11, 11-12
WaitEvent() 6-7, 7-4, 7-5, 13-10
Application mode 12-13
Application modes. 12-1, 12-10, 12-13
Default 3-20, 3-49, 12-9, 12-11, 13-3, 14-13
Starting in 10-13, 12-10
Arbitration order 5-9, 5-10, 14-17, 14-18

Arrival pattern 9-2, 14-4
Bursting clauses..... 14-5
Bursty..... 9-2, 14-4
Periodic 9-2, 14-4, 14-7
Planned..... 9-2, 14-4, 14-8
Arrival rates 9-2, 9-3, 14-4
Arrivalpoints 14-26
Implicit 11-22
Match 11-2
Next 11-2
Read-write 11-19
Tradeoffs..... 11-22
Arrived..... **11-1**
Asynchronous message passing 8-1
Auto-activated ...**4-19**, 11-21, 14-21, 14-27
Automatic vector table generation 5-5
Autostarting
Alarms 10-5, 10-13, 12-12
Tasks..... 12-11
Ticked schedules 11-12
Available targets **3-6**, **3-7**, 3-26

B

Basic conformance class 4-3
Basic tasks
About 4-2
BCC1 4-3
BCC2 4-3
ECC1 4-5
ECC2 4-5
Single-shot 4-2
States..... 4-3
Synchronization..... 4-23
Types 4-3
Waiting 4-24
BCC1..... 4-3
BCC2..... 4-3
Behavior
Concurrent..... 4-1
Looping..... 14-21, 14-22
Retriggering 14-21
Worst-case 14-23, 14-26
Best task priorities analysis15-1, 15-16, 15-21
Binding Manual 12-2, 12-3, 12-9
Blocking.....14-3, 15-6, 15-11
Times 13-11
Budgets
Execution timing..... 13-10
Missing overrun..... 13-11
Overrun..... 13-10
Timing..... 13-10
Buffer size **17-2**
Buffered
Activation..... 15-9
Buffered interrupts
Looping..... 14-21
Retriggering 14-21
Building
Custom 3-64
Bursting clauses..... 14-5
Bursty arrival pattern..... 9-2, 14-4
Busy period..... 15-11

C

Callbacks
Advanced schedule driver11-15, 11-16
Alarms..... 10-9
Cancel_<ScheduleID>..... 11-16
Canceling
Alarms..... 10-9
Categories..... 17-5
Category 1
Interrupt handlers..... 5-6
Interrupts 5-2
Category 2
Interrupt handlers..... 5-7, 5-12
Interrupts 5-2
OS level..... 5-4
CCCB 8-1
Chained activations 10-5
ChainTask()..... `4-8`, `4-18`, 4-25
ChainTaskset() ..... `4-8`, `4-18`, 4-25
ClearEvent()..... 7-4
Clearing
Events 7-4
Clock speed
Analysis..... 15-14

CloseCOM() 8-10
COM
About 8-1
Initializing 8-10
Shutting down 8-10
Starting 8-10
Stopping 8-10
COM options
#include file 8-3
Command line
Options 16-2
rtabuild 16-1
Using RTA 16-1
Compact IDs **17-1**
Compact Time **17-1**
Compile time error checking 13-14
Concurrent **4-1**, 4-25, 5-4, 6-6
Conformance classes
BCC1 & BCC2 ..... 2-2, 4-3
CCCA & CCCB 2-2
CCCB 8-1
ECC1 & ECC2 ..... 2-2, 4-5
Constants 10-11
Counter 10-12
Schedules 11-16
Context saving
Floating-point 15-4
Co-operatively scheduled 4-14
Counter/alarm mechanism. 10-1, 11-1, 11-22
Counters
About 10-1, 10-16
Attributes 10-10
Constants 10-12
Declaring 10-2
Incrementing 10-5
Units 10-11
Counters and alarms
Advanced concepts 10-9
CPU
Clock rate analysis .. 15-1, 15-21
Clock rate optimization ... 15-19
Time 10-13
Creating
Accessors 8-5, 8-6
Alarms 10-2
Counters 10-2
Events 7-1
Interrupts 5-5
ISRs 3-36
Messages 8-1
New applications 3-6, 3-26
Periodic schedules 11-3
Planned schedules 11-6
Receivers 8-4
Resources 6-1
Responses 3-28, 3-43, 9-1
Senders 8-4
Stimuli 3-28, 9-1
Tasks 4-6
Tasksets 4-19
Critical sections . **5-8**, 6-1, 6-8, 13-11
Custom builds 3-64
Cyclic alarms 10-1, 10-6, 10-8, 10-13

D

Data 12-4, 12-7
Data types
Messages 8-2
Deadlines
Analysis 15-16
Explicit 14-8
Implicit 14-7, 14-8
Responses 14-1, 14-8
Default interrupts **5-11**
Definitely
Schedulable 15-12
Unschedulable 15-12
Delays
Modifying 11-20
Development process
About 3-1
Building 3-5
Functional testing 3-5
Implementation 3-3
Specification 3-2
Timing analysis 3-6
Direct
Activation chains 4-21
DisableAllInterrupts() 5-8

Disabling Interrupts 5-8
Driver
Schedules..... 11-3
Driving a counter..... 10-1, 14-11

E

ECC1..... 4-5
ECC2..... 4-5
Embedding system **14-24**, 14-30
EnableAllInterrupts() 5-8
Enabling
Hook Routines..... 13-1
Interrupts 5-8
Timing measurement..... 13-9
Entry function **4-7**, **4-8**, 4-9, 14-12
ENUM.....17-7, 17-8
Enumerations 17-5
Error
Configuring..... 13-6
Handling & execution monitoring 13-1, 13-15
Handling at compile time 13-14
Hook..... 13-4
Logging..... 13-6
Using error logging 13-7
Error handling & execution monitoring
Advanced concepts 13-6
ETC 13-9
Events..... 7-1
Actions..... 10-10
Activate on message transmission 8-12
Activating tasks..... 4-18
Advanced concepts 7-5
Advanced setting 7-6
Clearing 7-4
Clearing multiple..... 7-5
Configuring..... 7-1
Idle Task..... 7-5
Masks 7-2
Setting . 7-1, 7-4, 7-5, 7-6, 8-12
Setting multiple..... 7-5
Setting with a message 7-6
Setting with an alarm..... 7-6
Synchronization points..... 7-1
Waiting 7-3
Waiting multiple..... 7-5
Execution
Measuring times..... 13-10
Monitoring 13-1, 13-15
Multiple profiles 14-20
Profile..... 3-38
Timing budgets 13-10
Execution ordering
Direct activation chains..... 4-21
Priority levels 4-22
Tasks..... 4-21
Execution time
Measuring and monitoring 13-9
Worst case 14-3, 14-10
Execution Time Correction..... 13-9
Expired..... **10-1**
Expiry
Alarm 10-13
Explicit
Activation..... 4-3
Response deadlines 14-8
Extended tasks.....4-2, 4-4, 14-29
States 4-4
Types..... 4-5
External stimuli 9-1

F

Fast activation..... 4-15
FIFO 3-54, 4-4, 8-11
Filters..... 17-5
Fixed priority.....2-1, 4-1, 4-22
Flags..... 8-14
Floating-point 4-16, 5-10
Context saving 15-4
Fully reentrant **3-71**
Functional testing phase 3-5

G

GetAlarm()..... 10-8, 10-13
GetAlarmBase()..... 10-11
GetArrivalpointDelay()..... 11-20
GetArrivalpointTasksetRef()..... 11-21
GetExecutionTime()..... 13-11

GetResource() .. 6-2, 6-3, 6-4, 6-5, 6-8, 6-10
Nesting 6-3
GetStackOffset() .. 13-12, 13-13, 13-14
GetStopwatch()3-7, 3-27, 13-9, 13-10, 13-11, 15-3, 15-4
GetStopwatchUncertainty()... 13-9, 13-10
Getting resources 5-9, 6-1, 6-2
Graphical display
Graphics tab..... 3-57, 10-3

H

Harvard 12-7
Heavyweight termination..... 4-8
Hex..... 12-9
Hook routines
Defining 13-1
Enabling 13-1
Hooks
Error..... 13-4
Macros..... 13-2
Mandatory 13-2
Optional..... 13-1, 13-2
Overrun..... 13-2
Shutdown 13-3
Stack fault..... 13-2, 13-5
Startup..... 13-3

I

Idle mechanism 3-17, 3-45, 4-12
Idle task
About 4-12
Events 7-5
Modeling 14-13
Immediate inheritance priority ceiling protocol..... 6-1, 6-5
Implementation of response 9-3, 14-3, 14-9
Implicit
Arrivalpoints..... 11-22
Deadlines 14-7
Response deadlines 14-8
Imprecise computation 13-15
Include file..... 8-3
Incrementing counters..... 10-5
Indeterminate
Blocking 15-11
Schedulability 15-7, 15-11
Indeterminately schedulable..... 15-12
Indirect
Activation..... 14-27
Indirectly activated stimuli..... 14-27
InitCOM()..... 8-10
Initialize 12-2
Initializing COM..... 8-10
Input jitter 14-25
Interface
Static..... 6-4
Interference 14-3, 15-6
Internal
Resources .. 3-21, 3-49, 3-54, 3-60, 3-62, 4-4, 4-12, 4-14, 6-6, 6-7, 6-8, 6-10, 15-3, 15-4, 15-16, 15-18, 15-22
Stimuli..... 9-1
Interrupt 12-5
Interrupt arbitration order.... 5-9, 5-10, 14-18
Interrupt handlers
About..... 5-6
Category 2 5-12
Interrupt lock times..... 14-14
Interrupt priorities..... 5-3
Interrupt priority level 5-3
Interrupts..... 12-5
About..... 5-1
Advanced concepts 5-9
Arbitration order .5-9, 5-10, 14-18
Category 1 5-2
Category 2 5-2
Configuring..... 5-5
Critical sections..... **5-8**
Default..... **5-11**
Disabling 5-8
Enabling 5-8
Floating-point..... 5-10
Handlers..... 5-1, 5-6
Interrupt lock times 14-14
OS level 5-4
Priority.....4-6, 5-3, 5-5

Response times 5-12
Resuming all..... 5-8
Resuming Category 2 5-8
Suspending all..... 5-8
Suspending Category 2 5-8
User level 5-4
Vectors..... 5-1, 5-5
Intervals..... 17-5
IPL 5-3, 12-5
ISRs
Accessors 8-4
Attaching to stimuli..... 3-41
Configuring..... 3-36
Creating..... 3-36
Execution profile 3-38
Floating-point..... 4-16
Multiple execution profiles .. 14-20
Overrun..... 15-15
WithCopy..... 8-7

J

Jitter
About 14-24
Input 14-25
Output 14-25

L

Lightweight termination 4-8
Linked resources..... 6-5
Linker 12-5, 12-8
Locking protocol
About 6-1
Priority ceiling protocol..... 6-1
Locking times 14-14
Looping..... 14-21

M

Macros 13-2
Mandatory hooks 13-2
MAXALLOWEDVALUE 10-10
Maximum response delay 14-25
Measuring & monitoring execution time..... 13-9
Measuring execution times..... 13-10
Memory..... 12-3, 12-5
Message passing
About..... 8-1
Asynchronous..... 8-1
Message resource 8-8
Messages
About..... 8-1
Accessors 8-4
Activating tasks 4-18, 8-12
Advanced concepts 8-11
Asynchronous..... 8-1
Configuring..... 8-1
Data transmission 8-1
Data type 8-2
Flags..... 8-14
Queued 8-11
Receivers 8-4
Receiving..... 8-8
Senders 8-4
Sending..... 8-8
Setting events..... 7-6, 8-12
Transmission mechanisms.... 8-6
Transmission mode..... 8-11
WithCopy..... 8-6, 8-11, 8-12
WithoutCopy..... 8-6, 8-12
MINCYCLE..... 10-11
Minimizing stack usage..... 15-4
Minimum response delay..... 14-25
Mixed-mode transmission 8-12
Modeling the application for the RTA-OSEK Planner
Advanced concepts 14-19
Modes 10-13, 12-1
Multi-level platforms..... 5-1
Multiple
Execution profiles 14-20
Mutual exclusion . 3-54, 4-4, 6-1, 6-10

N

Namespace..... 3-71
Nesting resource calls 6-3
New application..... 3-6, 3-26
Next values
Modifying..... 11-20
Non-preemptive tasks..... 4-14

Non-queued activation 15-8
Non-time based schedule units .. 11-16
Not schedulable..15-7, 15-8, 15-9, 15-15, 16-2
Now_<ScheduleID> 11-15

O

Offsets 10-6, 11-17, 11-18, 11-19, 13-14
    Periodic 11-17
OIL file
    Viewing 3-8, 3-27
Operating system
    Shutdown 12-1, 12-13
    Startup 12-1, 12-13
Options
    Command line 16-2
OS level 5-4, 12-5
OSEK 2-4
    COM 8-1
    Hooks 13-1, 13-2
osek_idle_task ..3-21, 3-49, 3-50, 4-12
OSErrorGetServiceId() 13-7, 13-8
Output files 16-4
Overrides
    Analysis **14-27**
Overrun 15-15
OverrunHook() 13-2, 13-10

P

Performing analysis..... 15-1, 15-21
Periodic
    Offsets 11-17
Periodic arrival pattern. 9-2, 14-4, 14-7
Periodic schedules 11-1
    Activating tasks 4-18
    Configuring 11-3
    Periodic stimuli 11-1
Pessimism
    Analysis 14-19
Planned arrival pattern. 9-2, 14-4, 14-8
Planned schedules 11-1, 14-26
    Activating tasks 4-18
    Analysis overrides **14-27**
    Aperiodic stimuli 11-1
    Configuring 11-6
    Modifying at run-time 11-19
    Modifying auto-activated tasks 11-21
    Modifying delays 11-20
    Modifying next values 11-20
Platforms
    Multi-level 5-1
    Single-level 5-1
Policy
    Scheduling **4-1**, 4-22
Predefined tasksets 4-20
Pre-emption control mechanism 6-8
Primary profiles 3-30, 3-39, 3-42, 3-43, 3-45, 9-1, 10-2, 11-3, 11-7, 14-11, 14-21, **14-25**, 14-30, 15-6
Priority
    Allocation ...4-23, 15-18, 15-19, 15-21, 15-22, 16-4
    Ceiling 6-1
    Fixed 2-1, 4-1, 4-22
    Interrupt 5-3
    Interrupts 4-6, 5-5
    levels 4-22
    OS level 5-4
    Required lower priority tasks 15-19
    User level 5-4
Priority ceiling protocol 6-1, 6-5
Priority level
    Interrupts 5-3
Processor utilization 15-10
Profiles
    Execution profiles 14-20
    Multiple execution profiles .. 14-20
    Primary. 3-30, 3-39, 3-42, 3-43, 3-45, 9-1, 10-2, 11-3, 11-7, 14-11, 14-21, **14-25**, 14-30, 15-6

Q

Queued messages
    About 8-11
    Destructive read 8-11
    Transmission mode 8-11

R

Race conditions 4-21, 6-1, 6-9
Avoiding 6-9
RAM 12-1, 12-2, 12-4, 12-5, 12-6, 12-7, 12-8
Minimizing usage 11-22
ReadFlag() 8-14
Read-write arrivalpoints 11-19
Real
Time 10-13
Real-time systems 14-1
Receivers
Accessors 8-4
Configuring 8-4
Transmission mode for queued messages 8-11
Receiving messages 8-8
Reentrancy 3-71
Relative alarm **10-1**, 10-7, 10-14
ReleaseResource() 6-2, 6-3, 6-4, 6-8, 6-10
Nesting 6-3
Releasing resources 6-1, 6-2
Renaming
Accessors 8-6
Required lower priority tasks 15-19
RES_SCHEDULER 3-54, 6-8, 6-9
Reset 12-1
ResetFlag() 8-14
Resource use times 14-14
Resources 5-9, 6-1, 6-4, 6-6, 6-10
Advanced concepts 6-8
Configuring 6-1
Getting 5-9, 6-1, 6-2
Internal .3-21, 3-49, 3-54, 3-60, 3-62, 4-4, 4-12, 4-14, 6-6, 6-7, 6-8, 6-10, 15-3, 15-4, 15-16, 15-18, 15-22
Linked 6-5
Locking protocol 6-1, 6-5
Message 8-8
Nesting calls 6-3
Releasing 6-1, 6-2
Resource use times 14-14
Scheduler 6-8
Response deadline 14-1, 14-8
Response delay
Maximum 14-25
Minimum 14-25
Response implementation 14-3
Response time **14-2**
Worst-case 2-1, 6-8, **14-2**, **14-3**, 14-13, 15-5
Response times 5-12
Responses 3-2, 9-1
Configuring 3-28, 9-1
Creating 3-43
Deadline 14-1, 14-8
Deadlines 14-9
Designing 9-3
Explicit deadlines 14-8
Generating 14-9
Implementation .9-3, **14-3**, 14-9
Implicit deadlines 14-8
Stimulus/Response model 9-1
Stimulus/Response model timing 14-4
Time **14-2**
Restarting single-shot schedules 11-13
ResumeAllInterrupts() ... 5-8, 8-14, 10-9
ResumeOSInterrupts() 5-8
Resuming
RTA-OSEK Component 12-10
Resuming all interrupts 5-8
Resuming Category 2 interrupts 5-8
Resuming tasks 4-1
Retriggering 14-21
Return codes 16-2
ROM 12-2, 12-5, 12-6, 12-7, 12-8
Round-robin scheduling 14-20
RTA
Command line 16-1
rtabuild 16-1
rtabuild 16-1, 16-2, 16-3, 16-4
Command line options 16-2
Messages 16-1
output files 16-4
Return codes 16-2
RTA-OSEK
Namespace 3-71

Reentrancy 3-71
RTA-OSEK Component . 2-1, 2-2, 12-1, 12-9, 12-10, 12-13
Hooks 13-1
Resuming 12-10
Shutting down 12-10
Starting 12-9
Starting in modes 12-10
Suspending 12-10
Tasksets 4-19, 4-21, 4-25
RTA-OSEK GUI 2-1
Development process 3-1
RTA-OSEK Planner .. 4-15, 10-14, 14-1, **14-2**, 14-3, 14-4, 14-8, 14-13, 14-15, 14-16, 14-17, 14-19, **14-27**, 14-28, 14-29, 15-1, 15-7, 15-8, 15-9, 15-10, 15-11, 15-12, 15-21, 16-2
Utilization 15-10
Run-time fault tolerance 13-15

S

Saving
Applications 3-8, 3-27
Schedulability analysis .. 3-57, 15-1, 15-4, 15-21
Schedulable ... **3-6**, 4-15, 14-13, 15-7, 15-8, 15-10, 15-11, 15-12, 15-15, 15-16, 15-19, 16-1
Schedule(). 3-54, 4-14, 4-15, 6-7, 14-4
Schedule/Arrivalpoint tradeoffs .. 11-22
Scheduler ... 4-1, 4-22, 6-8, 6-10, 13-3
Fixed priority 4-1
Schedules 3-42, 3-54, 4-14, 4-15, 6-7, 11-1, 11-17, 11-22, 14-4
Activating tasks 4-18
Advanced 11-3, 11-13
Advanced concepts 11-12
Advanced driver callbacks 11-15, 11-16
Analysis overrides **14-27**
Aperiodic stimuli 11-1
Arrived **11-1**
Autostarting ticked 11-12
Configuring periodic 11-3
Configuring planned 11-6
Constants 11-16
Driver 11-3
Minimizing RAM usage ... 11-22
Modifying auto-activated tasks 11-21
Modifying delays 11-20
Modifying next values 11-20
Non-time based units 11-16
Now 11-2
Periodic 11-1
Periodic offsets 11-17
Periodic stimuli 11-1
Planned 11-1, 14-26
Restarting single-shot 11-13
Schedules 11-12
Single-shot 14-28
Starting 11-10
State 11-2
Status variables 11-2
Ticked 11-3, 11-12
Ticking 11-11
Tradeoffs 11-22
Types 11-1
Unschedulable .. **3-6**, 15-7, 15-10, 15-12, 15-15
Worst-case behavior . 14-23, 14-26
Scheduling - round-robin 14-20
Scheduling policy **4-1**, 4-22
Fixed priority 4-1
Section 12-1, 12-6, 12-8
Senders
Accessors 8-4
Configuring 8-4
Transmission mode for queued messages 8-11
Sending messages 8-8
SendMessage() 13-8
Sensitivity analysis ... 3-59, 15-1, 15-12, 15-21
Clock speed 15-14
Deadlines 15-16
Overrun 15-15
Serially reentrant **3-71**
Set_<ScheduleID> 11-15
SetAbsAlarm() 10-13, 10-14
SetArrivalpointDelay() 11-20
SetEvent() 7-4, 7-6, 10-10

SetScheduleNext()..... 11-13, 11-21
Setting
Alarms 10-6
Events 7-1, 7-4, 8-12, 10-10
Shutdown..... 12-1, 12-13
COM 8-10
Hook 13-3
RTA-OSEK Component.... 12-10
ShutdownOS() 3-20, 3-21, 3-49, 12-10, 12-13, 13-3
Single-level platforms 5-1
Single-shot
Alarm..... **10-1**, 10-6, 10-15
Schedules..... 14-28
Schedules..... 11-13
Tasks..... 4-3
Specification phase..... 3-2
Stack depth analysis 15-1, 15-21
Floating-Point Context Saving 15-4
Stack fault hook 13-2, 13-5
Stack usage 13-12
Minimizing 15-4
Worst case ..14-10, 14-12, 15-3
StackFaultHook() 13-2, 13-5, 13-6
StartCOM()..... 8-10, 8-11
Starting
COM 8-10
RTA-OSEK Component..... 12-9
RTA-OSEK Component in modes..... 12-10
Schedules..... 11-10
StartOS.. 12-1, 12-2, 12-3, 12-5, 12-7, 12-9, 12-10, 12-11, 12-12, 12-13
StartOS() 3-20, 3-21, 3-49, 4-12, **4-19**, 5-8, 8-10, 10-5, 10-13, 11-12, 12-9, 12-10, 12-11, 12-12, 12-13, 13-14
StartSchedule() 11-10, 11-11
Startup 12-1, 12-3, 12-10, 12-13
Startup and shutdown..... 12-1, 12-13
Advanced concepts 12-10
Startup hook 13-3
State_<ScheduleID> 11-15
Static interface
About 6-4
ActivateTask_TaskID()..... 4-17
ChainTask_TaskID..... 4-18
ChainTaskset_TasketID() .. 4-18
Status variables
Match 11-2
Next 11-2
Now 11-2
State 11-2
Stimuli 3-2, 9-1
Activated profile 9-1
Aperiodic..... 11-1
Arrival patterns..... 9-2, 14-4
Arrival rates 9-2, 9-3, 14-4
Attaching ISRs 3-41
Configuring..... 3-28, 9-1
Designing 9-4
Direct activation 3-21, 3-49, 14-28
External 9-1
Indirect activation..... 14-27
Internal..... 9-1
Periodic 11-1
Primary profile 9-1
Stimulus/Response model 9-1
Stimulus/Response model timing 14-4
Visualizing the design..... 9-4
Stimulus/Response model 9-1
Timing..... 14-4
StopCOM() 8-10
Stopping
COM 8-10
Schedules..... 11-12
StopSchedule()..... 11-12
Stopwatch 13-9
Speed..... **3-7**, 3-27
Uncertainty..... 13-9
Summary
Counters and alarms 10-16
Error handling & execution monitoring 13-15
Events 7-6
Interrupts 5-12
Introduction to system modeling..... 9-4
Messages 8-15

Modeling the application for the RTA-OSEK Planner..... 14-29
Performing analysis 15-21
Resources..... 6-10
Schedules..... 11-22
Startup and Shutdown 12-13
Tasks..... 4-25
SuspendAllInterrupts() 5-8, 10-9
Suspending
All interrupts 5-8
Category 2 interrupts 5-8
RTA-OSEK Component.... 12-10
SuspendOSInterrupts()..... 5-8
Synchronization
Alarms 10-14, 12-13, 14-19
Basic tasks..... 4-23
Points..... 7-1
System
Modeling 9-1, 9-4
Timings 14-16

T

Target specific
Arbitration ordering 14-17
Interrupt recognition time 14-17
Timing information..... 14-16
Task switching
Scheduler ..4-1, 4-22, 6-8, 6-10, 13-3
Scheduling policy **4-1**, 4-22
Task Tracepoints..... 17-5
Tasks ... 3-5, 3-17, 3-19, 3-21, 3-43, 3-44, 3-45, 3-47, 3-50, 3-54, 3-55, 3-56, 3-58, 3-59, 3-60, 4-1, 4-7, **4-8**, 4-12, 4-21, 6-7, 13-4, 15-8, 15-9, 16-1, 16-4
Accessors 8-4
Activating on message transmission 8-12
Activation..... 4-11
Activation by alarm 4-18
Activation by event..... 4-18
Activation by message..... 4-18
Activation by periodic schedule 4-18
Activation by planned schedule 4-18
Activation counts.2-3, 4-11, 15-7
Advanced activation 4-17
Advanced concepts 4-13
Auto-activation..... 4-19
Autostarting 12-11
Basic..... 4-2
BCC1 4-3
BCC2 4-3
Characteristics..... 4-20
Configuration..... 4-6
Direct activation 4-17, 4-21
Direct activation chains..... 4-21
ECC1..... 4-5
ECC2..... 4-5
Entry function **4-7**, **4-8**, 4-9, 14-12
Execution ordering 4-21
Execution profile..... 3-38
Extended 4-2, 4-4, 14-29
Fast activation 4-15
Floating-point..... 4-16
Idle..... 4-12
Multiple activation..... 4-13
Multiple execution profiles .. 14-20
Non-preemptive 4-14
Overrun..... 15-15
Priority levels 4-22
Required lower priority 15-19
Resuming 4-1
Simulating waiting for basic tasks..... 4-24
Single-shot 4-3
Switching 4-1
Synchronization..... 4-23
Tasksets.....4-19, 4-21, 4-25
Terminating API..... 4-8
Termination..... 4-8
Upward activation 14-4
WithCopy..... 8-7
WithoutCopy..... **8-7**
Worst-case execution time 14-3
Tasksets.....4-19, 4-21, 4-25

Creating 4-19
Predefined 4-20
TerminateTask() .. 4–8, 4-23, 4-25, 6-9
Terminating
API **4-8**
Heavyweight 4-8
Lightweight 4-8
Tasks 4-8
Tick_<CounterID>() 10-5
Ticked schedules 11-3, 11-11, 11-12
Autostarting 11-12
TickSchedule_<ScheduleID> 11-11, 11-12
TICKSPERBASE 10-11
Time
CPU 10-13
Real 10-13
Timer/Counter hardware 3-21, 3-50
Times
Blocking 13-11
Timing
Analysis phase 3-6
Budgets 13-10
Measurement 13-9
Timing measurement
Enabling 13-9
Trace type **17-1**
Tracepoints 17-4
Tractable analysis 15-12
Tradeoffs
Schedules/Arrivalpoints 11-22
Transmission
About 8-6
Messages 8-11
Mixed-mode 8-12

U

Uncertainty 13-9
Units
Counter 10-11
Non-time based schedule 11-16
Unschedulable . **3-6**, 15-7, 15-10, 15-12, 15-15
Reasons 15-7
Upward activation 14-4
User level 5-4
Utilization
Processor 15-10

V

Variant **3-6**, 3-26
Vector table 5-1
Automatic generation 5-5
Vectors
About 5-1
Default interrupt **5-11**
Interrupt 5-5
Viewing
OIL file 3-8, 3-27
von Neumann 12-7

W

WaitEvent() 6-7, 7-4, 7-5, 13-10
Waiting
Basic tasks 4-24
Event 7-3
WithCopy 8-6, 8-11, 8-12
WithoutCopy 8-6, 8-12
Worst-case
Behavior 14-23, 14-26
Execution time 14-3, 14-10
Response time 2-1, 6-8, **14-2**, **14-3**, 14-13, 15-5
Stack usage . 14-10, 14-12, 15-3

Support

For product support, please contact your local ETAS representative.

Office locations and contact details can be found on the ETAS Group website www.etasgroup.com.